

Language-Parametric **Editor Services** from Language Specifications

DANIEL A.A. PELSMAEKER

Language-Parametric Editor Services from Language Specifications

Language-Parametric Editor Services from Language Specifications

Dissertation

for the purpose of obtaining the degree of doctor
at Delft University of Technology
by the authority of Rector Magnificus,
Prof. dr. ir. H. Bijl,
chair of the Board for Doctorates,
to be defended publicly on
Wednesday, 11 November 2026 at 17:30

by

Daniël Alexander Augustijn Pelsmaeker

This dissertation has been approved by the promotors.

Composition of the doctoral committee:

Rector Magnificus	*chairperson*
Prof. dr. A. E. Zaidman	Delft University of Technology, *promotor*
Dr. C. Bach	Delft University of Technology, and FORM, University of Southern Denmark, *promotor*

Independent members:	
Prof. dr. C. Omar	University of Michigan, U.S.A.
Dr. M. Madsen	Aarhus University, Denmark
Prof. dr. T. van der Storm	Centrum Wiskunde & Informatica (CWI), and University of Amsterdam
Prof. dr. A. van Deursen	Delft University of Technology
Dr. M. A. Migut	Delft University of Technology
Prof. dr. ir. F. A. Kuipers	Delft University of Technology, *reserve member*

Prof. dr. E. Visser (Delft University of Technology) was the original promotor and supervisor of this research until his untimely passing on April 5th, 2022.

The work in the dissertation has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics), and supported by the Dutch Research Council (NWO) VICI *Language Designer's Workbench* project (639.023.206).







Cover: Daniel A. A. Pelsmaeker
IPA Dissertation Series: 2026-19
Printed by: Gildeprint — https://gildeprint.nl

The author set this dissertation in LaTeX using the Libertinus and JetBrains Mono fonts.

Copyright © 2026 by Daniel A. A. Pelsmaeker — https://pelsmaeker.net

ISBN 978-94-6563-073-1

An electronic version of this dissertation is available at
`https://repository.tudelft.nl`

to Aya
who taught me yet another language

Contents

Summary **xi**

Preface **xix**

1 Introduction **1**
1.1 Editor Services . . . 2
1.2 Domain-Specific Programming Languages . . . 4
1.3 Developing Editor Services . . . 5
1.4 Deriving Static-Semantic Editor Services . . . 8
1.5 Contributions and Structure . . . 10

2 Towards Language-Parametric Semantic Editor Services **13**
2.1 Introduction . . . 14
2.2 Characterizing Editor Services . . . 16
2.3 Introduction to Statix . . . 17
2.4 Informing Editor Services . . . 20
2.5 Code Completion. . . . 22
2.6 Extract Definition . . . 26
2.7 Related Work. . . . 29
2.8 Conclusion . . . 31

3 Language-Parametric Static-Semantic Code Completion **35**
3.1 Introduction . . . 36
3.2 Code Completion. . . . 40
3.3 Background: Language Definition in SDF3 and Statix . . . 42
3.4 Constraint-Based Code Completion . . . 46
3.5 Code Completion Algorithm . . . 53
3.6 Evaluation . . . 61
3.7 Related Work. . . . 66
3.8 Future Work . . . 71
3.9 Conclusion . . . 72

4 Language-Parametric Reference Synthesis 75
4.1 Introduction . . . 76
4.2 Scope Graphs and Statix . . . 80
4.3 Reference Synthesis by Example . . . 85
4.4 Operational Semantics . . . 89
4.5 Heuristics . . . 98
4.6 Experimental Evaluation . . . 102
4.7 Related Work . . . 105
4.8 Conclusion . . . 106
4.9 Future Work . . . 106
4.10 Data Availability . . . 108

5 Conclusion 111
5.1 Related Work . . . 112
5.2 Discussion . . . 113
5.3 Future Work . . . 116

Appendices 118

A Language-Parametric Static-Semantic Code Completion 119
A.1 Syntax Definition of Micro-C# in SDF3 . . . 119
A.2 Static Semantics of Micro-C# in Statix . . . 121
A.3 Syntax Definition of Tiger in SDF3 . . . 125
A.4 Static Semantics of Tiger in Statix . . . 128
A.5 Definitions of Strategy Combinators . . . 134

B Language-Parametric Reference Synthesis 137
B.1 Soundness Proofs . . . 137

Bibliography 147

Abbreviations 163

Curriculum Vitæ 169

List of Publications 171

Dissertation Series 173

Summary

Designing a programming language from scratch gives the language developer complete freedom to define and implement the syntax and semantics of their language. To aid the user in the understanding, navigation, and production of code in the new language, its development environment needs *editor services* such as syntax highlighting, code completion, type checking, code navigation, debugging, and refactoring. These editor services can be implemented by hand, an approach taken by many IDEs because it allows their developers to optimize for performance and tailor the implementation to their editor and platform. However, establishing the correctness of these editor services can be difficult. In particular, refactorings, which should preserve program behavior, have been shown in certain cases to introduce compile-time errors, or even worse, silently change the behavior of the program.

To avoid much of this labor and to improve the soundness of the editor services, we argue for automatically deriving these editor services from a declarative specification of the language. To do this, we use the Spoofax language workbench, which allows specifying the grammar and static semantics of a programming language. From these specifications, Spoofax already derives a syntax highlighter, type checker, and basic code navigation. But there are other editor services that we would like to derive in a language-parametric manner, such as refactorings. This leads to our research question:

> How can we derive editor services from the static-semantic specification of a programming language?

To answer this question, we followed a *design-science* methodology: we explored the problem and designed, implemented, and evaluated possible solutions.

As the static semantics of languages in the Spoofax language workbench are specified using constraints, we explored how to express editor services as constraint-solving problems. We proposed a blueprint for how find usages/references, code completion, and the extract-definition refactoring could be expressed as constraint problems that can be solved using a specialized constraint solver.

Then we applied our blueprint to implement a language-parametric code completion editor service that is sound and complete with respect to the language's static-semantic

specification. It proposes syntax, identifiers, and function calls that are semantically valid at the program's caret position.

Furthermore, we implemented a building block for binding-preserving program transformations: a language-parametric and sound-by-construction reference synthesis function. The function synthesizes references by inverting the name-lookup functions of the underlying language, with the required name-resolution information automatically derived from the static-semantic specification of the language.

We conclude that semantic editor services can be derived from programming language specifications, and demonstrate that such services can be implemented in a language-parametric and sound-by-construction manner. A formal specification of the syntax and semantics of a programming language can therefore serve not only as a definition of the language, but also as an input to language-parametric tests, tools, and editor services. This shifts most of the implementation effort from language-specific editor services to reusable language-parametric tooling.

Samenvatting

Het vanaf nul ontwerpen van een programmeertaal geeft de taalontwikkelaar volledige vrijheid om de syntax en semantiek van de taal te definiëren en te implementeren. Om de gebruiker te ondersteunen bij het begrijpen, navigeren en produceren van code in de nieuwe taal, heeft de ontwikkelomgeving *editorvoorzieningen* nodig, zoals syntaxkleuring, codeaanvulling, typecontrole, codenavigatie, foutopsporing en herstructurering. Deze editorvoorzieningen kunnen handmatig worden geïmplementeerd, een aanpak die in veel geïntegreerde ontwikkelomgevingen wordt gevolgd omdat zij daarmee de prestaties kunnen optimaliseren en de implementatie kunnen afstemmen op hun editor en platform. Het kan echter lastig zijn om de correctheid van deze editorvoorzieningen aan te tonen. Met name herstructureringen, die het programmagedrag zouden moeten behouden, kunnen in bepaalde gevallen compileerfouten introduceren of, nog erger, ongemerkt het gedrag van het programma veranderen.

Om een groot deel van dit werk te vermijden en de correctheid van de editorvoorzieningen te verbeteren, pleiten wij ervoor editorvoorzieningen automatisch af te leiden uit de declaratieve specificatie van de taal. Hiervoor gebruiken we de Spoofax taalwerkbank, waarmee de grammatica en statische semantiek van een programmeertaal kunnen worden gespecificeerd. Uit deze specificaties leidt Spoofax al een syntaxkleurder, typecontroleur en basale codenavigatie af. Er zijn echter ook andere editorvoorzieningen die we op een taalparametrische manier zouden willen afleiden, zoals herstructureringen. Dit leidt tot onze onderzoeksvraag:

> Hoe kunnen we editorvoorzieningen afleiden uit de statisch-semantische specificatie van een programmeertaal?

Om deze vraag te beantwoorden, hebben we de *design-science*-methodologie gevolgd: we onderzochten het probleem en ontwierpen, implementeerden en evalueerden mogelijke oplossingen.

Omdat de statische semantiek van talen in de Spoofax taalwerkbank door middel van beperkingen (constraints) wordt gespecificeerd, hebben we onderzocht hoe editorvoorzieningen kunnen worden uitgedrukt als beperkingsoplossingsproblemen. We stelden een blauwdruk voor waarin het vinden van gebruikslocaties/verwijzingen, codeaanvulling en

de definitie-extractieherstructurering kunnen worden uitgedrukt als beperkingsproblemen die door een gespecialiseerde beperkingsoplosser kunnen worden opgelost.

Vervolgens hebben we onze blauwdruk toegepast om een taalparametrische codeaanvullingseditorvoorziening te implementeren die correct en volledig is ten opzichte van de statisch-semantische taalspecificatie. Deze voorziening stelt syntax, identificatoren en functieaanroepen voor die semantisch geldig zijn op de cursorpositie in het programma.

Daarnaast hebben we een bouwsteen geïmplementeerd voor bindingbehoudende programmatransformaties: een taalparametrische en door constructie correcte functie voor verwijzingssynthese. Deze functie synthetiseert verwijzingen door de naamopzoekfuncties van de onderliggende taal om te keren, waarbij de benodigde informatie over naamresolutie automatisch wordt afgeleid uit de statisch-semantische specificatie van de taal.

We concluderen dat semantische editorvoorzieningen kunnen worden afgeleid uit programmeertaalspecificaties en tonen aan dat zulke voorzieningen taalparametrisch en door constructie correct kunnen worden geïmplementeerd. Een formele specificatie van de syntax en semantiek van een programmeertaal kan dus niet alleen dienen als definitie van de taal, maar ook als invoer voor taalparametrische tests, hulpmiddelen en editorvoorzieningen. Dit verschuift het grootste deel van de implementatie-inspanning van taalspecifieke editorvoorzieningen naar herbruikbare taalparametrische hulpmiddelen.

Preface

I have always enjoyed creating things that are useful to other people. This combination of creativity and usefulness has guided me throughout my life. As a child, I decided I wanted to become an *inventor*, and even coined a name for the institute I would one day found: *Daniël's Uitvinders Instituut* (DUI). That institute has yet to materialize, however.

After high school, I studied Architecture at Hanze University of Applied Sciences in Groningen, where I obtained my bachelor's degree, and then continued to pursue a master's degree in Architecture at Delft University of Technology.

Somewhere along the way, however, I realized that the culture and world of architecture did not quite resonate with me. I had always been drawn to the principle that *form follows function*, but during my master's, I increasingly encountered design for the sake of design. Eventually, I realized that architecture was not the right path for me.

Having already completed my bachelor's degree, I had the opportunity to leave the Architecture master's program and start studying Computer Science at Delft University of Technology instead. So I decided to pivot.

Although I had taken a detour through architecture, computers had always felt natural to me. My first encounter with computers, as the story goes, involved me crawling around the house unplugging the computer my mother was working on. I like to think I was simply demonstrating that turning it off and on again solves many problems.

Years later, I wanted to play games on our Commodore 64, but I was still too young to read. My mother would write the necessary commands on a sticky note, and I would type them in by matching the characters on the keyboard. Sometimes, she would show me some basic programming, dazzling young me with my name rapidly scrolling across the screen:

```
10 PRINT "HALLO DANIEL"
20 GOTO 10
RUN
```

That was the beginning of my journey with programming. Every few years, a newer computer would enter the house, with the older one handed down to me. That is how I eventually got to experiment with QBASIC in DOS, and later with Visual Basic on Windows 3.0, which my father also used for work. For many years, everything I wrote was, in one way or another, *basic*.

Eventually, I learned C#, which introduced me to the curly-brace family of languages that includes C, C++, and Java. Around the same time, I also started exploring lower-level programming and tried to make my own operating system. Written partly in C and partly in assembly, it eventually managed to boot from a floppy disk and display something resembling a DOS prompt. Just a blinking cursor and some built-in commands, but I was very pleased with the result.

After completing my bachelor's degree in Computer Science, I continued with a master's program, during which Eelco Visser's lectures on compiler construction piqued my interest in programming languages. Until then, programming languages had always seemed like ageless tools: things that already existed, built long ago by brilliant people, and that the rest of us simply used. Discovering that programming languages themselves could be designed, studied, and improved opened an entirely new world to me. Their syntax and semantics were not fixed; they were things I could define. It felt a bit like constructing a natural language — except that this one actually had to work.

I am very grateful that, after completing my master's thesis in the Programming Languages group under Eelco's supervision, he offered me a PhD position in his group, which I gladly accepted. Eelco was an ambitious and inspiring researcher, and he taught me a great deal about academic writing and conducting research. I am thankful for the opportunity he gave me, and for the time I had to get to know and learn from him.

I want to thank Andy for stepping in, for his proactive support, and for helping me focus on what mattered most to bring this dissertation to completion. I am very grateful to Casper for being my supervisor and collaborator throughout these years. I have always enjoyed our productive discussions, through which we both gained new insights and deepened our understanding of the research. Your expertise, particularly in formal systems, complements my own, and I sincerely hope we continue to find opportunities to work together.

It was a pleasure to work with my collaborators Aron and Hendrik. Your knowledge and enthusiasm were instrumental in producing high-quality publications and bringing them across the finish line. I am also grateful to my independent committee members, Cyrus, Magnus, Tijs, Arie, and Gosia, for taking the time to read my dissertation, provide feedback, and be part of my defense.

My PhD was a special six-year program in which, alongside conducting programming language research, I worked as a software developer in the Computer Science and Engineering Teaching team in Delft. I want to thank the members of the Programming Languages research group for creating such a kind, supportive, and productive environment. I am equally grateful to the dedicated team of developers and educators in the teaching team for their camaraderie and for providing me with a welcome change of pace whenever my re-

search motivation was running low. In the teaching team, I worked with Danny, Max, and Elmer to improve the WebLab learning environment for students. Working with you was a blast, and I learned a lot about maintaining software, servers, and infrastructure.

Beyond work, I am grateful for the many friendships that made these six years so enjoyable. Thank you, Luka, Cas, Jaro, Andreea, Azqa, Feyza, Houda, and Eduardo, for always being up for role-playing revelry, an Egyptian escapade, a culinary quest, or just a casual conversation. I have fond memories of the fun times we shared.

A special thank you to my dear friends Gabriël, Jeff, Ruben, Danny, and Mozhan. Thank you for the frequent adventures we share, whether around a board game, in a virtual world, or out and about together. Our friendship has been a source of joy for so many years.

Another special thank you goes to Sára, Jelco, and Beky for the countless conversations, games, and wonderful times together. Thank you for your support, and for welcoming me into your lives and making me feel like part of your family. Your warmth and friendship means a great deal to me.

My family, of course, deserves special thanks. Thank you to my siblings, Sebastiaan and Fé, for being the loving, supportive, and hardworking people I can only aspire to be. I admire your dedication and the care you put into everything you do, and I am so fortunate to have you both in my life. I hope you know how much I appreciate you.

To my parents, Alex and Claudia, thank you for your unconditional love and support, not just throughout these six years, but throughout my life. You have always encouraged me to pursue my interests and channel my creativity. While the inventor's institute I imagined as a child never came to be, the desire to build remained. In the end, it led me to become an architect after all: a *software architect*. Your love was a necessary condition for me to finish this PhD and finally add another title: *doctor*.

Last but not least, to my dear wife Aya. We started this PhD journey together — who would have thought that an introductory graduate school course could introduce me to the most amazing person in my life? These six years, through the COVID pandemic and the frustrations of a PhD, have brought us so much more than a dissertation. We have shared so many adventures, experiences, and milestones, and *inshallah* we will share many, many more. Throughout all these years, your care, patience, and support have meant more to me than I can express. God knows how grateful I am that we get to close this chapter together, and even more so that we get to begin the next one side by side. Thank you, Aya.

Daniel
Delft, November 2026

1

Introduction

To some, his long beard might seem an indication of his wisdom. To him, it is merely a distraction, as is having a social life. Solitude is his mana, bugs his jam. While the tome on his workbench is merely a reference — he mastered arcane COBOL long ago — his specialized implements are more useful. If he wields them correctly, he can be sure his transformations are sound and complete. Murmuring to himself, he conjures up the last imperative symbols. He knows, of course, that in general it is undecidable whether any incantation works as expected. Gödel proved this. But the glowing symbols give him confidence. With a final flick of the wrist, the contraption of metal and wheels springs to life, shining its light upon its creator. He bound its name; it will do his bidding.

To the layperson, programming can seem "magical" and the expert programmer like a wizard wielding their power with ease. As Brooks (1995) remarks, "The magic of myth and legend has come true in our time. One types the correct incantation on a keyboard, and a display screen comes to life, showing things that never were nor could be." And while not exactly magical, programming is indeed a difficult skill to master. It requires analytical thinking, problem-solving, creativity, and a deep knowledge of the programming language and programming environment. A computer is inherently an exact machine, and consequently, programmers have to be very precise in how they write their code, or it will be rejected.

1

To manage the cognitive overhead of programming, most programmers use Integrated Development Environments (IDEs) that provide a coding environment with editor services for each supported programming language (Stack Overflow, 2025). Editor services improve program comprehension by giving the programmer insight into the semantics of the program and tools to navigate and modify the code (Kosar, Mernik, and Carver, 2011; Maalej et al., 2014; Siegmund, 2016). Their goal is to reduce the effort and cognitive burden of reading and writing code and improve its quality. Common editor services include syntax highlighting, (semantic) code completion, live type checking, codebase navigation, debugging, and refactoring tools.

Implementing editor services is a major effort, as they are highly specific to each programming language. As a result, while there are a lot of programming languages, editors often support advanced editor services only for the most popular (mainstream) languages (JetBrains, 2025; Microsoft, 2025a; Microsoft, 2025b). Arguably, there is a relation between a language's popularity and the editor services that exist to support developing in said language. As editor services are useful, we want more programming languages to have advanced editor services. In this dissertation, we explore ways to reduce this effort of providing editor services for new programming languages.

1.1 Editor Services

Many problems are more easily solved with the aid of a computer, but it is difficult to express the problem in terms that a machine can process. Instead, we use programming languages as a bridge between the problem domain and the machine domain. Programming languages allow the programmer to focus on expressing the problem using higher-level abstractions, hiding lower-level complexities, such as those of the underlying machine (Bal and Grune, 1994; Brooks, 1996).

Programming is a complex activity that requires careful attention to detail and a deep understanding of the programming language, the problem domain, and the codebase itself. To support this effort, programmers read and write their code in an editor. Editors range from pure text editors (e.g., Vim, Emacs, SublimeText) to sophisticated Integrated Development Environments (IDEs, e.g., IntelliJ IDEA, VS Code, Eclipse) that provide advanced features to support software development (Juhár and Vokorokos, 2015). These environments offer *editor services* that assist programmers in understanding, navigating, and managing code, to highlight its semantics and reduce the likelihood of errors.

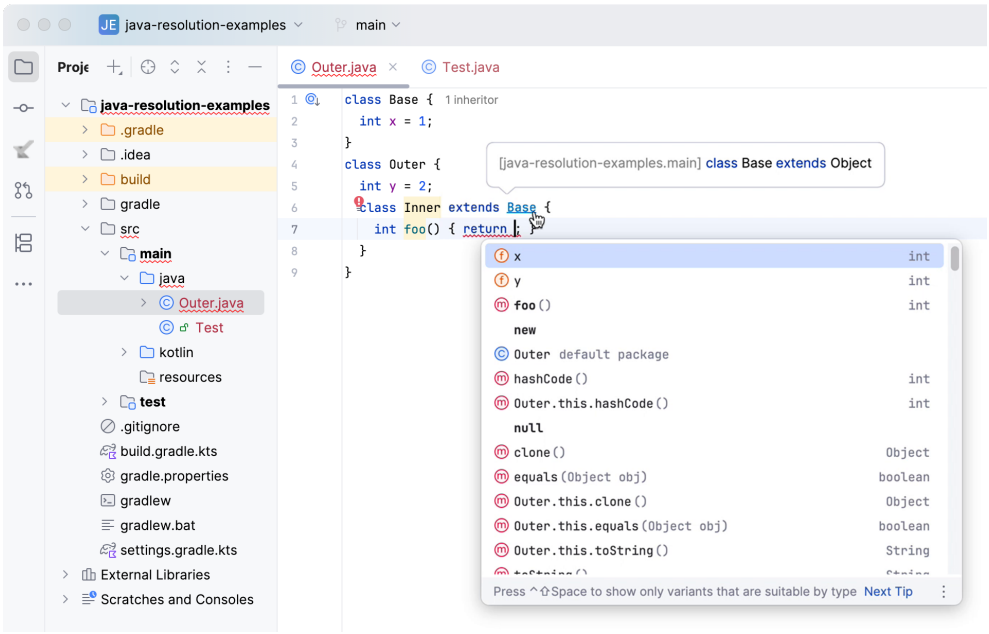


Figure 1.1: The IntelliJ IDEA development environment showing syntax highlighting, code completion, warning and error markers, hover type information, a quick-fix button, among other editor services. The user is about to navigate to the `Base` class definition by clicking its usage in the code.

An *editor service* is any tool that aids the understanding, navigation, or production of code in a programming language.[1] Editor services streamline the coding processes by offering features that help users read, write, edit, and manage code more effectively. Editor services:

- improve readability and understanding by formatting code and highlighting syntactic and semantic information and errors;
- provide tools to navigate and organize the codebase; and
- reduce the coding effort and improve its quality by providing code suggestions and automated transformations and refactorings.

As such, editor services are very useful, and most programmers depend on them. Figure 1.1 shows syntax highlighting, code completion, and hover information among other editor services in IntelliJ IDEA.

[1] *Editor service* as a term has been used in previous works (Keidel, Pfeiffer, and Erdweg, 2016; Erdweg, Kats, et al., 2011). Although we could not find an official definition of *editor service*, for the purposes of this dissertation, we will follow this definition.

Not all programming languages are well-supported by editors, as editor services are typically only implemented for the most popular programming languages. While only a handful of programming languages are regularly used in various applications (Stack Overflow, 2025), there are thousands of programming languages (Pigott and Axtens, 2020). A substantial number of these languages are *domain-specific* programming languages.

1.2 Domain-Specific Programming Languages

In our daily lives we communicate amongst ourselves, both in speech and writing, using a natural language such as English or Dutch. These natural languages are extremely versatile, allowing the expression of many different ideas across a vast array of disciplines. This makes them *general-purpose* natural languages. Conversely, on the road, you may encounter a traffic controller who uses hand signals to communicate to you whether you must stop, continue, or move somewhere else. At the hospital, you find doctors and nurses use medical names and shorthands to efficiently communicate about a condition and how to treat it. These are examples of *domain-specific* natural languages, where people communicate using the idioms and concepts of a particular domain of expertise. While it might be hard for you as a layperson to understand what they are talking about, the use of idioms and jargon allows them to communicate effectively and efficiently using well-defined domain-specific concepts and terminology, including the nuances that might be hard to explain to an outsider. The domain-specific language lowers the communication barrier between people of similar professions and removes ambiguity.

In programming, we can similarly broadly categorize each programming language according to the domain in which it expresses its concepts. If this domain is very broad, the language is usable in many different problem domains while not specific to any of them. This allows the programming language to be used for application development, machine learning, or to implement games. We call such a language a *general-purpose programming language* (GPL). Examples include Python, Java, Rust, and JavaScript.

Conversely, Fowler (2011) defines a *domain-specific programming language* (DSL) as a computer programming language of limited expressiveness focused on a particular domain. DSLs are programming languages with a higher level of abstraction, optimized to express solutions in the idiom and concepts of the problem domain. This allows people fluent in the DSL's domain to use the language, even if they are not software people. There are many different DSLs, each fit for a specific task or domain. For instance, LaTeX (Lamport, 1989) was used to write and typeset this dissertation, to be printed on a digital printing system configured using CSX (Denkers et al., 2021), whose hardware chips might be described in VHDL (IEEE, 1987).

DSLs can be designed to have a domain-specific syntax and provide domain-specific checks and error messages. And because DSLs abstract over the underlying complexities and boilerplate code, whole classes of domain-specific bugs are prevented by making it impossible to express these in the DSL. For example, LaTeX avoids many typesetting issues, CSX avoids creating incompatible printer configurations, and VHDL prevents timing issues in hardware designs. This allows domain experts to understand, develop, modify, and validate solutions in the domain they are familiar with. DSLs reduce the communication overhead between the domain experts and programmers, tame accidental complexity, and have been attributed to improvements in code quality and maintainability, and programmer productivity (Fowler, 2011; van Deursen, Klint, and Visser, 2000; Mernik, Heering, and Sloane, 2005). As a result, it can make sense to develop a new DSL for a specific problem domain, and to invent a DSL to solve a particular problem is a technique known as *language-oriented programming* (Ward, 1994).[2]

A big downside of using a DSL, however, can be the lack of support for the language in an editor. Editor services that provide tools such as syntax highlighting, code completion, and type checking are available for most mainstream languages, but may not be available for the DSL.

1.3 Developing Editor Services

Editor services are not only language-specific but also tightly integrated into a specific editor or IDE. For example, in their IntelliJ IDEA development environment, JetBrains has implemented separate type checkers for each supported language, with the goal to support most editor services. However, due to implementation differences, IntelliJ IDEA supports for each language a different subset of editor services (JetBrains, 2010; JetBrains, 2018). Furthermore, there are often cases in which these editor services are *unsound*. A simple example of an unsound refactoring of a Java program in IntelliJ IDEA is shown in Fig. 1.2, where an automatic Rename refactoring results in code that behaves differently. The engineering effort of developing editor services that are sound is already high, and has to be repeated for every editor that should support the language.

Embedded versus External DSLs DSLs can be broadly categorized as either *embedded DSLs* or *external DSLs* (Fowler, 2011). An embedded DSL[3] is developed, deployed, and used within an existing general-purpose host programming language. This lets the DSL provide domain-specific primitives by exploiting the available extension options of the host

[2] "A popular definition of a computer scientist is one who, when presented with a problem to solve, invents a new language instead!" (Katzen, 2001)

[3] Embedded DSLs are also known as *internal DSLs*.

```
1 class Base {
2   int x = 1;
3 }
4 class Outer {
5   int y = 2;
6   class Inner extends Base {
7     int foo() { return y; }  // foo() == 2
8   }
9 }
```

(a) Before automatic renaming.

```
1 class Base {
2   int y = 1;                 // x → y
3 }
4 class Outer {
5   int y = 2;
6   class Inner extends Base {
7     int foo() { return y; }  // foo() ≠ 2
8   }
9 }
```

(b) After automatic renaming.

Figure 1.2: Using IntelliJ IDEA to apply the automatic RENAME refactoring of field `x` to `y` on line 2 in the Java code, causes the reference to `y` on line 7 to no longer resolve to the `y` field on line 5. IntelliJ IDEA should have re-qualified the reference on line 7 to be `Outer.this.y` for the refactoring to be behavior-preserving.

language, while using the host language's existing syntax, semantics, and runtime environment. Some GPLs are more suitable to be extended than others. Popular embedded DSLs include LINQ in C# (Meijer, 2011), the HTML embedded DSL in Kotlin (Aigner et al., 2024), and SQLAlchemy Core in Python (Bayer, 2012).

Embedding a DSL in a host language reduces the effort of designing, implementing, and maintaining the DSL, as it can leverage the existing host language's tooling and editor services, such as syntax highlighting, code completion, and code navigation. However, the embedded language cannot change the host language's syntax or semantics. Therefore the host language's tooling does not understand the DSL-specific syntax, semantics, concepts, or domain, and cannot provide DSL-specific editor services. Similarly, the host language compiler is limited in its ability to check domain-specific invariants or verify correct usage of the DSL at compile time. This can create a *leaky abstraction* (Spolsky, 2004), where the underlying host language's intricacies and limitations peek through, or where the host language permits incorrect usage of the DSL. As a result, the host language's editor services can be unsound or incomplete with respect to the DSL, with no way for the language developer to improve this.

Instead, in this dissertation, we focus on *external* DSLs: those DSLs that are developed as standalone languages. External DSLs give the language developer complete freedom to define and implement the syntax and semantics of their language to any level of detail. However, as there is no underlying host language to build upon, they also have to expend the extra effort to develop high-quality editor services specific to their language. To avoid much of this labor, we argue for an alternative approach where we *generate* these editor services from a declarative specification of the language.

Declarative Meta-DSLs To specify the syntax and semantics of the new language, a language developer can use a DSL to specify DSLs: a so-called *meta-DSL*. For example, instead of implementing a parser specifying *how* to parse the language, a language developer can declaratively describe the grammar of the language using, say, Yacc (Johnson, 1975), Bison (Corbett, 1985), or ANTLR (Parr, Dietz, and Cohen, 1992). This allows the developer to express the syntax using the Backus–Naur form notation (BNF; Backus et al., 1960) commonly used to specify grammars, and then use the corresponding program to *generate* an implementation from this declarative specification. This both gives a declarative specification of an aspect of the language (e.g., syntax), and ensures the resulting parser matches the specification exactly.

As Strachey (1973) remarks: "There are so many programming languages in existence that it is a hopeless task to attempt to learn them all. Moreover, many programming languages are very badly described; in some cases, the syntax, or rather *most* of the syntax, is clearly and concisely defined, but the vitally important question of the semantics is almost always dealt with inadequately." Therefore, in addition to specifying the syntax, we also need a way to specify the semantics of a programming language and integrate the language into an IDE. This can be done using a language workbench.

Language Workbenches While standalone meta-DSLs are often used to generate parts of the compiler or interpreter for a language (such as the parser), ideally we also want a good integration of the language into an editor or IDE. This integration is provided by a *language workbench*, a term coined by Fowler (2005) for a development environment that facilitates the design, implementation, deployment, and use of DSLs. Language workbenches include JetBrains MPS (Völter, 2011), MontiCore (Krahn, Rumpe, and Völkel, 2010), Rascal (Klint, van der Storm, and Vinju, 2009), Xtext (Eysholdt and Behrens, 2010), DrRacket (Flatt, 2012), SugarJ (Erdweg, Rendel, et al., 2011), and Spoofax (Kats and Visser, 2010b; Visser, Wachsmuth, et al., 2014). A comparison of language workbenches is given by Erdweg, van der Storm, Völter, Boersma, et al. (2013) and Erdweg, van der Storm, Völter, Tratt, et al. (2015). In this dissertation, we focus specifically on the Spoofax language workbench.

Spoofax Language Workbench Positioned as the *Language Designer's Workbench* (Visser, Wachsmuth, et al., 2014), the Spoofax language workbench lets the language developer write a declarative specification of the syntax and semantics of their language. From this specification, both tooling (parsing, type checking) and editor services (syntax highlighting, pretty-printing, code navigation) for the new language are automatically derived. This is shown in Fig. 1.3. Spoofax allows the developer to focus as much as possible on the domain com-

1

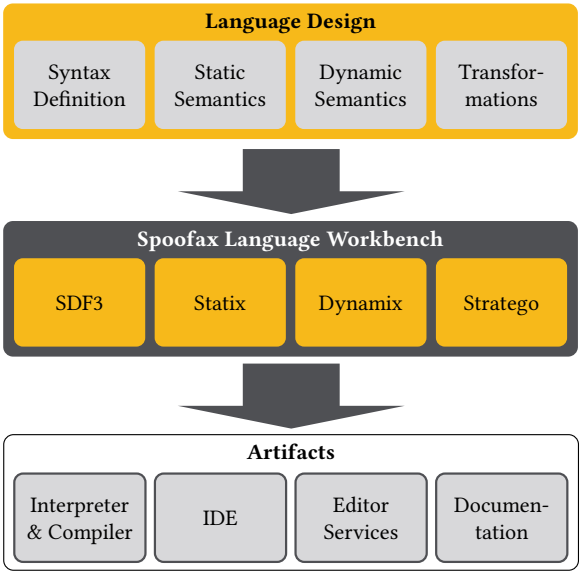


Figure 1.3: High-level idea of the Spoofax *Language Designer's Workbench* (Visser, Wachsmuth, et al., 2014), where aspects of the language are defined declaratively using meta-DSLs from which multiple artifacts are automatically derived.

plexity and DSL design, while reducing the cost of implementing editor services by deriving their implementations automatically from the language specification.

The specification of a DSL in Spoofax is done across several meta-DSLs, such as SDF3 for the syntax (Vollebregt, Kats, and Visser, 2012), Statix for the static semantics (van Antwerpen, Bach, et al., 2018b), and Dynamix for the dynamic semantics (Molendijk, 2022). This removes the need for up-front development and foresight, making it possible to derive editor services after-the-fact with minimal to no changes to the language definition, while providing a unified experience for users of DSLs developed using Spoofax.

1.4 Deriving Static-Semantic Editor Services

Static semantics describe the properties of the program that can be determined without actually evaluating or running the program (Møller and Schwartzbach, 2020). The static semantics typically include the *types* of variables, functions, and operations, such as whether they are a number, text (string), or boolean, and the *scopes* in which these variables and functions are visible. A type checker is then used to infer these properties and verify that they are not violated, such as ensuring that the program never tries to add a number to a string of text. In the Spoofax language workbench, the static semantics of a language are specified

declaratively using the Statix meta-DSL (van Antwerpen, Bach, et al., 2018b). A language specification in Statix is given as predicates on terms, types, and a *scope graph* (van Antwerpen, Néron, et al., 2016), from which Spoofax automatically derives constraint-based type checking and type inference. Currently, the only editor services Spoofax derives from the static semantics are static-semantic type inference and checking, and basic code navigation.

By interpreting the Statix declarative specification of static semantics in novel ways, we hypothesize that it will be possible to derive other editor services in a language-parametric manner, such as refactorings. This leads to the main research question of this dissertation:

> How can we derive editor services from the static-semantic specification of a programming language?

The resulting editor services are positioned in a design space whose dimensions determine both the guarantees they provide and their practical usefulness. In particular, relevant dimensions include an editor service's *soundness*, that is, whether every result it produces is guaranteed to satisfy the static-semantic specification; its *completeness*, that is, whether it can produce every result that satisfies that specification; and its *performance*, since editor services are typically used interactively and must therefore respond quickly. These dimensions are not independent and therefore may give rise to trade-offs: stronger soundness guarantees or more complete results can require more computation and thereby negatively impact performance. As discussed earlier, IDE developers may therefore deliberately accept unsoundness in uncommon cases, if doing so improves performance and usability.

To answer the research question, we conducted our research by exploring the problem and by designing, implementing, and evaluating possible solutions, a method common in programming language research and congruent with the *design science* paradigm (Hevner et al., 2004; Engström et al., 2020). We first explored the problem of deriving editor services from static-semantic specifications, and constructed a blueprint for solutions that express editor services as constraint-solving problems. Then we implemented two solutions in the context of the Spoofax language workbench: a language-parametric code completion editor service and a building block for binding-preserving program transformations.

Spoofax had already been the test bed for other language-parametric approaches related to the implementation of parsing and analysis of programming languages. Thus Spoofax gave us a jumping-off point that avoided most of the implementation overhead that a standalone implementation would have had. Furthermore, we could leverage the languages with existing static-semantic specifications in the Statix meta-language. However, Statix and the existing specifications were focused on deriving type checkers. Our approach to derive editor services from these existing specifications forced us to deal with the hidden assumptions

and corner cases that such specifications have, moving our research from a purely theoretical realm into the actual reality, where every detail must be accounted for. This also gave us and others early opportunities to experiment with our approach, and produced language-parametric artifacts that would be useful to future users of Spoofax.

Finally, we evaluated our approach on programming languages for which we had the Statix static-semantic specification and a large number of existing tests. To verify our implementations, we used the existing tests of each language to derive tests that evaluate the working of our implementations.

1.5 Contributions and Structure

This dissertation comprises three main chapters, each based on a peer-reviewed and accepted publication. To allow independent reading, the original publications are largely as published, with only some textual revisions and adaptations to the layout and typesetting to fit this dissertation's presentation style. These chapters make the following contributions:

- In Chapter 2 we express editor services using Statix constraints on placeholders (holes in the program), and identify the requirements imposed on the operational semantics of a Statix constraint solver that can solve these constraints.

  The chapter corresponds to the paper *Towards Language-Parametric Semantic Editor Services Based on Declarative Type System Specifications (Brave New Idea Paper)* (Daniel A. A. Pelsmaeker, Hendrik v. Antwerpen, and Eelco Visser, 2019), published in the proceedings of the *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. DOI: `10.4230/lipics.ecoop.2019.26`

- In Chapter 3 we implement a language-parametric code completion editor service by applying a specialized search strategy to the static-semantic constraints of a language, such that it returns only the syntax and names that are semantically valid at the caret position in a program.

  Chapter 3 corresponds to the paper *Language-Parametric Static Semantic Code Completion* (Daniel A. A. Pelsmaeker, Hendrik v. Antwerpen, Casper Bach, and Eelco Visser, 2022), presented at the *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2022)* and published in the OOPSLA1 issue of the *Proceedings of the ACM on Programming Languages (PACMPL)*, volume 6. DOI: `10.1145/3527329`

- In Chapter 4 we use the Statix static-semantic specification of a language to implement a language-parametric reference synthesizer function that can be used as part of a program transformation to ensure bindings are preserved or fixed.

  The chapter corresponds to the paper *Language-Parametric Reference Synthesis* (Daniel A. A. Pelsmaeker, Aron Zwaan, Casper Bach, and Arjan J. Mooij, 2025), presented at *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2025)*, where it won a Distinguished Artifact Award. It was published in the OOPSLA1 issue of *Proceedings of the ACM on Programming Languages (PACMPL)*, volume 9. DOI: `10.1145/3720481`

  For this publication, the author of this dissertation is shared first author with Aron Zwaan (with equal contributions to writing and implementation).

Finally, in Chapter 5 we answer the main research question of this dissertation and discuss the limitations and future work.

2

Towards Language-Parametric Semantic Editor Services based on Declarative Type System Specifications

In the previous chapter, we introduced the idea of deriving editor services from static-semantic specifications to reduce the development effort of a language and improve its adoption. This chapter categorizes editor services and gives a high-level idea on how we can work toward deriving these editor services using the constraints from a language specification in the Statix static-semantic specification language. This chapter is a slightly revised version of the Brave New Idea paper previously published as:

> Daniel A. A. Pelsmaeker, Hendrik van Antwerpen, and Eelco Visser (2019). "Towards Language-Parametric Semantic Editor Services Based on Declarative Type System Specifications (Brave New Idea Paper)." In: *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom*. Ed. by Alastair F. Donaldson. Vol. 134. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. DOI: 10.4230/lipics.ecoop.2019.26

Abstract

Editor services assist programmers to write and comprehend code more effectively. Implementing editor services correctly is not trivial. This chapter focuses on the specification of semantic editor services, those that use the semantic model of a program. The specification of refactorings is a common subject of study, but many other semantic editor services have received little attention. We propose a language-parametric approach to the definition of semantic editor services, using a declarative specification of the static semantics of the programming language and constraint solving. Editor services are specified as constraint problems, and language specifications are used to ensure correctness. We describe our approach for the following semantic editor services: reference resolution, find usages, goto subclasses, code completion, and the extract definition refactoring. We do this in the context of Statix, a constraint language for the specification of type systems. We investigate the specification of editor services in terms of Statix constraints, and the requirements these impose on a suitable solver.

2.1 Introduction

Editor services, such as syntax highlighting, reference navigation, and variable renaming, are an important tool for programmers. For example, code navigation is important for effective code comprehension (Robillard, Coelho, and Murphy, 2004), and refactoring approaches rely heavily on robust tool support (Mens, Demeyer, et al., 2003). It is therefore no surprise that such services are regularly used by users of IDEs (Murphy, Kersten, and Findlater, 2006).

Editor services can be classified as syntactic and semantic editor services. The former, such as syntax highlighting, rely only on the (abstract) syntax of a program. The latter depend on a semantic model of the program, and use type or name-binding information. Semantic editor services can be further divided into two groups: services that *inform* about the program, and services that *transform* the program. Informing services depend on the program model that is the result of type checking. The program model contains information on the types of variables, the declarations that references refer to, etc. Transforming services are guided by the program model (e.g., to rename a declaration and all its usages), but may also rely on the typing rules to ensure the resulting, transformed, program is well-formed.

Implementing semantic editor services and ensuring their correctness is not trivial (see, e.g., the difficulties around correctly implementing Java refactorings (Schaefer and de Moor, 2010)). Language workbenches aid the development of programming languages and environments (Fowler, 2005) by means of declarative formalisms and reusable tools that support correctness and reduce development effort. A good example is to use a context-free grammar to specify syntax. This specification can be used to drive a parser, but also for unparsing, or to provide syntactic code completion. The language developer writes a declarative spec-

ification, which helps with the correctness of the syntax, while existing parsing, unparsing, and code completion algorithms can be reused, reducing development time.

However, even though "editor support is a central pillar of language workbenches" (Erdweg, van der Storm, Völter, Boersma, et al., 2013), and many language workbenches do indeed support many common editor services, there is little literature on reusable formalisms and algorithms for their definition (Omar, Voysey, Hilton, Sunshine, et al., 2017). An important exception is the extensive work on defining correct refactorings (e.g., Schaefer and de Moor, 2010; Tip et al., 2011; Steimann, 2018). However, many editor services common to modern IDEs, such as reference resolution, finding declaration usages, or semantic code completion, have received little attention.

In this chapter, we argue that a range of semantic editor services, beyond those that have already appeared in the literature, can be specified as constraint problems. Constraints separate the declarative specification of a problem from the operational interpretations necessary to solve it. This separates concerns, but also allows reuse of constraint-based specifications for different purposes. For example, in addition to verifying the correctness of the static semantics of a program, constraint-based typing rules have also been successfully used in the implementation of semantically correct refactorings (Steimann, 2018).

Many editor services rely on name-binding information, where complex scoping and name-binding rules can be a challenge for the correct implementation of editor services (e.g., correct Java refactorings involving names (Schafer et al., 2012)). Although constraint-based formulations of typing rules are pervasive; constraint-based formulations of the scoping and name-binding rules are rare. Name binding introduces complexities, such as avoiding accidental name capture when refactorings introduce new names. We believe that treating name binding and name resolution as an integral part of the constraint problem increases the applicability of a constraint-based approach to editor services, and can improve existing specifications from the literature in this regard.

As the basis for our investigations we use Statix, a constraint language developed for the specification of type systems (van Antwerpen, Bach, et al., 2018b). Statix is built around scope graphs, a language-independent model for name binding and name resolution (van Antwerpen, Néron, et al., 2016). We argue that Statix is a suitable basis for the definition of editor services by expressing them in terms of Statix constraints and Statix type system specifications. Although Statix constraints are suitable for a declarative specification of editor services, the current deterministic solver algorithm of Statix, suitable for type checking and code navigation, is not capable of solving the editor scenarios we discuss. We identify requirements for an alternative solver for Statix that does support the interpretation and solving algorithms required for our proposed editor service definitions.

Specifically, we have the following contributions:

- We express several common editor services in terms of Statix constraint problems.
- We identify requirements for an operational semantics of Statix that can solve these problems.

This chapter is organized as follows. Section 2.2 discusses the characteristics of semantic editor services and motivates our choice of editor services. In Section 2.3, we introduce Statix, and Statix type specifications using an example. In Section 2.4 we express several informing editor services in terms of the resulting program model. In Section 2.5 and Section 2.6, we do the same for the semantic code completion and extract definition refactoring editor services, respectively. In Section 2.7 we discuss related work. We conclude and discuss future work necessary to fully realize our proposed approach in Section 2.8.

2.2 Characterizing Editor Services

Editor services can be characterized as syntactic, those that only need the syntactic model of the program to work, and semantic, those that require the semantic model of the program (Erdweg, van der Storm, Völter, Boersma, et al., 2013). We can further distinguish the semantic editor services by whether they transform the program, or merely inform the user. The informing services include editor services such as goto declaration, finding and highlighting usages, navigating to the supertype, and listing all overriding methods. The transforming editor services include quick fixes, static-semantics-preserving refactorings, and (semantic) code completion.

In this section, we discuss aspects that distinguish the various semantic editor services, and motivate our choice for the editor services we discuss.

Completeness Some editor services must be able to handle syntactically and/or semantically incomplete programs. For example, as code completion can be invoked while the user is typing, it must be able to deal with a program that is both syntactically and semantically incomplete. Similarly, the *fix import* quick fix that adds an import statement to a program to make a reference resolve, must be able to deal with programs that have incomplete semantic information; namely, the program with the reference that initially does not resolve. Other editor services could provide a better user experience if they could deal with syntactically or semantically incomplete programs, but this is not a requirement.

Preserving Static Semantics The transforming editor services all need to preserve the existing semantics of the program up to some degree. Refactorings such as *rename refactoring* and *extract definition* tend to have very strict semantic preservation requirements,

including that all existing references need to resolve to the same declarations before and after the refactoring. Quick fixes and code completion, by their nature, introduce new syntax that may change certain local semantics of the code, but should not have an impact outside the area of influence.

Concrete Name Generation Often, transforming editor services add new declarations to the program as part of their refactoring or fixing behavior. These declarations need a concrete name, one that is syntactically valid, does not clash with existing names in the program, and does not cause inadvertent variable capture.

What We Study Given the characterization above, we picked five editor services for which we describe the ideas of this chapter. As informing editor services we choose *reference resolution*, *find usages*, and *list subclasses*, because they show how scope graphs can be used to answer these queries, and the last one requires language-specific knowledge. The last two also explore how flexible the solver must be to answer such inverse queries.

We discuss two transforming editor services: *code completion*, which will have to deal with syntactically incomplete programs, and the *extract definition* refactoring, which is interesting because it introduces new syntax for which we want to use the solver to find the concrete, semantically correct, values to fill in.

We do not claim that these editor services cover all issues, or that the resulting requirements cover all editor services. However, we think that they exhibit a sufficient range of features to show the range of possibilities, and expose important requirements that need to be fulfilled to realize our approach.

2.3 Introduction to Statix

Statix is a recently introduced meta-language for the specification of static semantics (van Antwerpen, Bach, et al., 2018b), based on scope graphs and constraints (Neron et al., 2015; van Antwerpen, Néron, et al., 2016). We chose Statix because it allows us to declare semantic editor services in terms of constraints and type system specifications.

First, we explain scope graphs, a language-independent model for name binding and name resolution. Then, we introduce the rules for static semantics, and their (declarative) meaning. Finally, we explain how type checking based on these rules is implemented. We use the Java program in Fig. 2.2 as a running example. The subscripts on program identifiers are a notational convention we use to distinguish different occurrences of the same name.

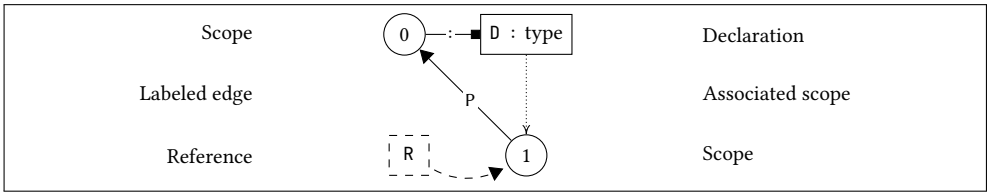


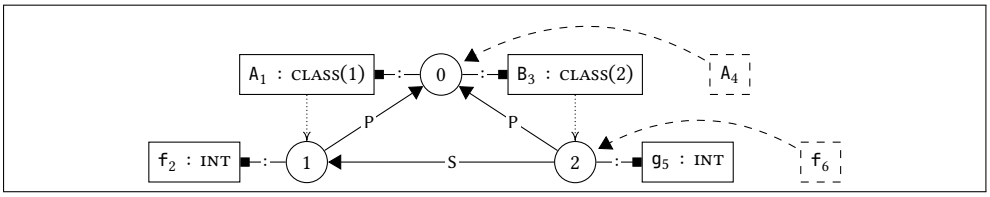
Figure 2.1: Overview of the notation used for scope graphs in this chapter.

Name Binding with Scope Graphs In Statix the name binding and resolution is part of the constraint problem, to allow complex interactions between type checking and name resolution. The name-binding structure of a program is represented as a language-independent model called a *scope graph* (Neron et al., 2015; van Antwerpen, Néron, et al., 2016; van Antwerpen, Bach, et al., 2018b), which is a graph of scopes and declarations in those scopes. As shown in Fig. 2.1, the scopes are connected by labeled, directed edges. Name resolution corresponds to a query finding a path in the graph to a matching declaration.

Consider our example program and the corresponding scope graph in Fig. 2.2. The global scope of the whole program is represented by the circled node 0. The definition of class `A` corresponds to a declaration $\texttt{A}_1$. Declarations contain both the name and its type, and therefore use the '*is of type*'-symbol "`:`" to label these edges. Class types are represented by the class scope. For example, scope 1 is the scope of class `A`, and its type is CLASS(1). The class scopes are lexical sub-scopes of the global scope, which is modeled by the `P`-labeled (parent) edges. The fact that class `B` extends class `A` is represented by the edge labeled `S` (supertype). This edge makes the fields from the super class visible in the subclass, but is also used to decide subtyping between class types. The field declarations are similar to the class declarations, but in the class scopes.

Resolving a name corresponds to querying the scope graph for a matching declaration. Resolution queries are parameterized by a regular expression that determines which declarations can be reached, and a predicate determining which declarations match. An additional order on labels is used to disambiguate multiple matching declarations. For example, the class reference $\texttt{A}_4$ is resolved in the global scope 0. Class references are resolved in the lexical context, and the regular expression that encodes this is $\texttt{P}^*\texttt{:}$, which matches any path to a declaration via any number of `P`-steps to lexical parents. The declaration itself should match the reference, which is specified with the predicate DECL($\texttt{A}_4$), which holds for any x_i where $x = \texttt{A}$. In this case, the reference resolves directly to declaration $\texttt{A}_1$ in scope 0.

```
class A1 {
    int f2 = -1;
}
class B3 extends A4 {
    int g5 = f6;
}
```



$$\text{(J-ClassDec)}\ \frac{\nabla s_c \quad s_c \xrightarrow{P} s \quad s \overset{:}{\longrightarrow} C_i : \text{CLASS}(s_c) \quad s_c \vdash \overline{d}\ \text{OK} \quad \text{query}\ s \xmapsto{P^*:} \text{DECL}(D_j)\ \text{as}\ D_k : \text{CLASS}(s_d) \quad s_c \xrightarrow{S} s_d}{s \vdash \texttt{class}\ C_i\ \texttt{extends}\ D_j\ \{\ \overline{d}\ \}\ \text{OK}}$$

$$\text{(J-FieldDec)}\ \frac{s \vdash [\![t]\!] \Rightarrow T \quad s \overset{:}{\longrightarrow} f_j : T \quad s \vdash e : T' \quad \vdash T' <: T}{s \vdash t\ f_j = e;\ \text{OK}} \qquad \text{(J-Var)}\ \frac{\text{query}\ s \xmapsto{P^*S^*:} \text{DECL}(x_i)\ \text{as}\ x_j : T}{s \vdash x_i : T}$$

$$\text{(T-Int)}\ \frac{}{s \vdash [\![\texttt{int}]\!] \Rightarrow \text{INT}} \qquad \text{(T-Class)}\ \frac{\text{query}\ s \xmapsto{P^*:} \text{DECL}(C_i)\ \text{as}\ C_j : T}{s \vdash [\![C_i]\!] \Rightarrow T}$$

$$\text{(<:-Int)}\ \frac{}{\vdash \text{INT} <: \text{INT}} \qquad \text{(<:-Class)}\ \frac{\text{query}\ s_1 \xmapsto{S^*} \text{SCOPE}(s_2)\ \text{as}\ p}{\vdash \text{CLASS}(s_1) <: \text{CLASS}(s_2)}$$

Figure 2.2: Example Java program with two classes, corresponding scope graph, and relevant Statix typing rules.

Resolving the variable reference $\texttt{f}_6$ follows the same pattern. However, it should be possible to resolve not just to variables in the lexical context, but also to fields in the super class. This is achieved by using the regular expression $\text{P}^*\text{S}^*:$. This allows the reference to be resolved to declaration $\texttt{f}_2$, by following the S-edge to scope 1.

Type Specifications The rules of a Statix specification formally describe the scope graph that corresponds to a program, as well as constraints on references and types, in terms of syntax-directed rules. Figure 2.2 shows some of the rules that apply to our example program. For example, the rule (J-ClassDec) specifies that a class definition c is well-formed in scope s, written as $s \vdash c$ OK, if the scope graph has the correct structure, and the definitions in the class are well-formed as well ($s_c \vdash \overline{d}$ OK). The first three premises state that the scope graph contains a scope s_c that is unique to this class (∇s_c), that this scope has a P-edge to

its lexical parent ($s_c \xrightarrow{\mathrm{P}} s$), and that there is a declaration C_i for the class in the lexical scope s, typed by the class scope s_c ($s \xrightarrow{\vdots} C_i : \textsc{class}(s_c)$). The last two premises say that the reference to the super class resolves to a declaration D_k, which is typed by a class scope s_d (query $s \xmapsto{\mathrm{P}^*\vdots} \textsc{decl}(D_j)$ as $D_k : \textsc{class}(s_d)$), and that an inheritance edge exists from the scope of this class to the scope of the super class ($s_c \xrightarrow{\mathrm{S}} s_d$).

The rule (J-FieldDec) specifies that a field declaration is well-formed if a declaration for the field exists in the scope graph ($s \xrightarrow{\vdots} f_j : T$), if the assigned expression is well-typed for some type T' ($s \vdash e : T'$), and the expression type T' is a subtype ($\vdash T' <: T$) of the semantic type T corresponding to the type annotation ($s \vdash [\![t]\!] \Rightarrow T$). The relations for semantic typing, subtyping, and expression typing are also defined with Statix rules. The only built-in constraints are to define the scope graph, to query the scope graph, and term equality. All other relations are completely determined by the rules from the specification.

Type Checking The specification is declarative, and only gives a logical description of what well-formed programs are with respect to a scope graph. We made no assumptions yet on how to operationalize it. One possible interpretation is to use the specification to type check programs. Checking that a program p is well-formed corresponds to checking if the constraint $s \vdash p$ OK is satisfiable. Van Antwerpen, Bach, et al. (2018b) describe an algorithm to solve such constraints, given a specification and a program p as input. The algorithm uses the rules from the specification to simplify constraints until only built-in constraints remain. These are solved using unification and scope graph resolution algorithms. This solver is deterministic: it does not use backtracking, and only applies rules if they match the given program construct. The result of solving a constraint such as $s \vdash p$ OK is a solution consisting of a variable assignment V and a scope graph G, or no solution if the constraint cannot be satisfied. A resulting program model would also include the types assigned to all expressions, and the resolution relation R mapping each reference to its declaration.

2.4 Informing Editor Services

Many editors offer editor services through which users can navigate their program. The simplest of these involve clicking a reference and jumping to the corresponding declaration, or listing all usages of a declaration, but there are also more sophisticated editor services such as those that list the subclasses of a particular class. All these services can be expressed as queries on the program model that resulted from type checking. Even though these queries themselves do not change the program, they may be part of the implementation of other editor services that do change the program. For example, a refactoring that renames a variable first needs to find all usages of the variable to ensure they are all renamed.

Reference resolution and finding declaration usages can easily be derived from the program model, which contains the resolution relation R, which consists of pairs of references and their declaration. Consider the example in Fig. 2.2 again. Finding the declaration corresponding to reference $\mathtt{A}_4$, involves finding the entry for the reference in R. Conversely, finding all usages of declaration $\mathtt{f}_2$ corresponds to a reverse lookup. These queries parallel the resolution queries in the typing rules, and can directly be derived from the specification.

While a query to find all subclasses of a certain class is not directly present in the typing rules, we can phrase such a query as a constraint, which we solve with respect to the given program model. For instance, how would we specify — in constraints — the query to get all subclasses of class $\mathtt{A}_1$? We assume as input the declaration itself, and the scope 0 of the class definition, which should be part of the program model. The general idea of the query is to find the class scope, find other class scopes that are connected to it by inheritance edges, and find their corresponding declarations. This is encoded by the following constraint:

$$\text{query } 0 \overset{:}{\longmapsto} \textsc{decl}(\mathtt{A}_1) \text{ as } \mathtt{A}_1 \ : \ \textsc{class}(s_c)$$
$$\text{query } s_d \overset{\mathrm{S}^+}{\longmapsto} \textsc{scope}(s_c)$$
$$\text{query } s' \overset{:}{\longmapsto} \textsc{true} \text{ as } x_i \ : \ \textsc{class}(s_d)$$

where s_c, s_d and s' are existentially quantified, and x_i is the output. The first constraint says that the class declaration is typed by a scope s_c. The second constraint states that there is a path from some subclass scope s_d to the class scope s_c. The final constraint indicates that there is a declaration with any name x_i, which is typed by the subclass scope s_d.

None of these constraints appear as such in the typing rules, and we have to do work to find possible solutions. This may seem daunting, given the free variables for scopes and names, both of which have infinite domains. However, we are only interested in solutions that are valid in the context of an existing scope graph. This scope graph is always finite, which gives us an initial, if maybe inefficient, strategy to find possible solutions. In the case of our example, there is one solution, where $s_c = 1$, $s_d = 2$, $s' = 0$, and $x_i = \mathtt{B}_3$.

The given formulation requires an algorithm quite different from the current, deterministic solver of Statix. Instead of strictly relying on inference via forward resolution and unification, it needs to be able to guess values, try different alternatives, and backtrack on failed attempts. An alternative approach could have been to change scope graph queries to allow backward edge steps. For example, if we use $\hat{l}$ for backward steps in the regular expression, our second constraint might have been

$$\text{query } s_c \overset{\hat{\mathrm{S}}^+}{\longmapsto} \textsc{true} \text{ as } s_d$$

In this case, we could do forward resolution from scope s_c again, reusing the resolution algorithm that is already there. Although this approach may work for queries designed specifically with editor services in mind, it does not work if we want to use our typing rules as-is. Therefore, we choose not to change the formalism, but require a solver that supports more flexible inference.

Summary We showed that queries on the program model can be expressed as constraints, and that finding answers to these queries corresponds to solving these constraints in the context of a given program model. We discussed that solving these queries requires different solver strategies to be supported by the solver for Statix. However, this solver would be independent of the specific object language the query is for, and is therefore reusable between languages. Given such a solver, implementing such editor services reduces to being able to specify the query as a constraint.

2.5 Code Completion

Code completion is an editor service that suggests a valid code fragment to be inserted at the caret position. This assists the user while typing, attempts to minimize typing errors, and aids in discovery by showing the possible syntax and references. Syntactic code completion is the most basic kind of code completion: it suggests only syntax fragments that fit at the caret location, with no regard for whether the proposal fits semantically. Semantic code completion improves on this by suggesting only those proposals that conform to the static semantics of the language, such as only suggesting expression syntax that can produce a value of the expected type. Additionally, semantic code completion proposes inserting references to declarations, such as variables, fields, and functions, that are visible from the scope at the caret location. In this section, we discuss how the type system and semantic specification of a language can be used to provide accurate semantic code completion without additional work on the part of the language designer.

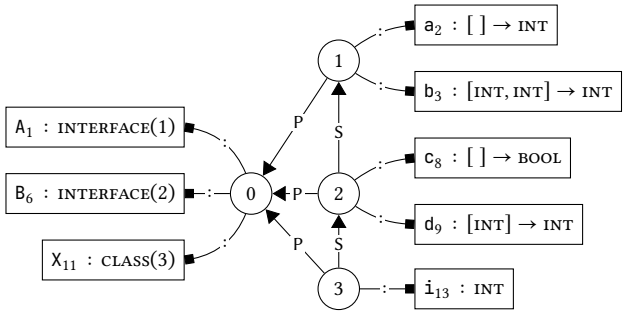
In Fig. 2.3, we show an example Java program with the caret position denoted by |, near the end of the last line of class $\mathtt{X}_{11}$. The program is incomplete: it is not syntactically valid because the user has not yet finished typing. Despite this, we would want the semantic model of the program so we can suggest relevant syntax and references.

As a first step, we propose to use the techniques described by de Souza Amorim, Erdweg, et al. (2016) to use the syntactic specification of the language to introduce *placeholders* into the abstract syntax. A placeholder is a term in the syntax that represents a place where syntax of a certain sort, such as an expression or a declaration, could be inserted. This makes the program syntactically complete, and the placeholders provide us with syntax terms that we can constrain. Therefore, to the completion service, the incomplete line of

```
interface A1 {
    int a2();
    int b3(int x4, int y5);
}

interface B6 extends A7 {
    boolean c8();
    int d9(int x10);
}

class X11 implements B12 {
    int i13 = |;
}
```



$$\text{(J-InterfaceDec)}\ \frac{\forall s_c \quad s_c \xrightarrow{P} s \quad s \xrightarrow{:} C_i : \text{INTERFACE}(s_c) \quad s_c \xrightarrow{S} \overline{s_d} \quad \text{query } s \xrightarrow{:} \text{DECL}(\overline{D_j}) \text{ as } \{\overline{D_k} : \text{INTERFACE}(\overline{s_d})\} \quad s_c \vdash \overline{d}\ \text{OK}}{s \vdash \texttt{interface}\ C_i\ \texttt{extends}\ \overline{D_j}\ \{\ \overline{d}\ \}\ \text{OK}}$$

$$\text{(J-InterfaceMethodDec)}\ \frac{s \vdash [\![C_i]\!] \Rightarrow T \quad s \vdash [\![\overline{C_k}]\!] \Rightarrow \overline{T_k} \quad s \xrightarrow{:} x_j : \overline{T_k} \to T}{s \vdash C_i\ x_j(\overline{C_k}\ \overline{x_k});\ \text{OK}}$$

$$\text{(J-ClassDec)}\ \frac{\forall s_c \quad s_c \xrightarrow{P} s \quad s \xrightarrow{:} C_i : \text{CLASS}(s_c) \quad s_c \vdash \overline{d}\ \text{OK} \quad \text{query } s \xrightarrow{:} \text{DECL}(\overline{D_j}) \text{ as } \{\overline{D_k} : \text{INTERFACE}(\overline{s_d})\} \quad s_c \xrightarrow{S} \overline{s_d}}{s \vdash \texttt{class}\ C_i\ \texttt{implements}\ \overline{D_j}\ \{\ \overline{d}\ \}\ \text{OK}}$$

$$\text{(J-FieldDec)}\ \frac{s \vdash [\![t]\!] \Rightarrow T \quad s \xrightarrow{:} f_j : T \quad s \vdash e : T' \quad \vdash T' <: T}{s \vdash t\ f_j = e;\ \text{OK}}$$

$$\text{(J-ThisMethodCall)}\ \frac{s \vdash \overline{e} : \overline{V} \quad \vdash \overline{V} <: \overline{U} \quad \text{query } s \xrightarrow{S^* :} \text{DECL}(m_i) \text{ as } \{m_j : \overline{U} \to T\}}{s \vdash m_i(\overline{e}) : T}$$

$$\text{(J-Plus)}\ \frac{s \vdash e_1 : T_1 \quad s \vdash e_2 : T_2 \quad T_1 = T_2 = T = \text{INT}}{s \vdash e_1 + e_2 : T}$$

Figure 2.3: Java program illustrating code completion, with its scope graph and relevant typing rules.

code has the following syntax, with placeholder *$Exp* for a possible expression that would complete the program:

```
int i13 = ($Exp);
```


At this point, we would want to invoke the solver and let it verify our program using the rules shown in Fig. 2.3. However, no rules apply to the placeholder term *$Exp*. Instead, we propose to replace any occurrence of a placeholder in the syntax terms with a corresponding constraint variable in the constraint terms. In this example, we use ε for *$Exp*, which, because of the semantic rule (J-FieldDec), results in the following constraints for this line:

$$3 \vdash [\![\texttt{int}]\!] \Rightarrow T \qquad 3 \dashrightarrow \texttt{i}_{13} : T \qquad 3 \vdash \varepsilon : T' \qquad \vdash T' <: T$$

Solving these constraints assigns $T' \mapsto \textsc{int}$ and $T \mapsto \textsc{int}$. In other words, the editor service has inferred that the expected type of the expression on that line must be INT, and produced the scope graph shown in Fig. 2.3. The solver can continue, trying to find an assignment for ε. There are two rules in Fig. 2.3 that it could apply: (J-Plus) and (J-ThisMethodCall). In fact, we would want the solver to return both solutions for code completion. We will explore both these alternatives.

Expression Completion From rule (J-Plus) ($s \vdash e_1 + e_2 : T$) we would get the assignment $\varepsilon = \varepsilon_1 + \varepsilon_2$, where ε_1 and ε_2 are new constraint variables introduced by the solver. We want to stop here, and let the solver return the solution $\varepsilon = \varepsilon_1 + \varepsilon_2$. Note that this solution is incomplete: it does not describe the whole program, as there are still free constraint variables in it. Therefore, the solver would need to be able to return incomplete solutions. As part of this solution, we get some constraints that not ground because they contain these free constraint variables:

$$3 \vdash \varepsilon_1 : \textsc{int} \qquad\qquad 3 \vdash \varepsilon_2 : \textsc{int}$$

Translated back to syntax terms, replacing the free constraint variables by placeholders, this would result in the following syntax on the line being completed:

```
int i13 = ($Exp) + ($Exp);
```

Of course, we could also let the solver continue its search to find assignments for ε_1 and ε_2, but this would likely result in an ever-expanding sequence of $\varepsilon_1 + \varepsilon_2 + \varepsilon_3 + \ldots$. Ultimately, there are infinitely many solutions if we were to try to make all variables ground. This shows that we need a way to instruct the solver on how deep we want a constraint variable to be solved. In this example, we want solutions for ε only one level deep.

Method Call Completion When the solver instead applies rule (J-ThisMethodCall), we get method call completion: where code completion suggests calls to methods in scope at the caret position, and whose return type is compatible with the expected type of the expression.

From rule (J-ThisMethodCall) ($s \vdash m_i(\overline{e}) : T$) we would get the assignment $\varepsilon = \mu(\overline{\varepsilon})$, again introducing new constraint variables μ and $\overline{\varepsilon}$ to represent the method name and arguments, respectively.

Since proposing just the syntax for a method call is not very satisfactory to a user, this time we *do* want to get another level of solutions. At least, we want μ to be solved, but we do not care about $\overline{\varepsilon}$. We need a way to indicate this to the solver.

Through the rule (J-ThisMethodCall) the solver would add these constraints:

$$3 \vdash \overline{\varepsilon} : \overline{V} \qquad \vdash \overline{V} <: \overline{U} \qquad \text{query } 3 \overset{S^*:}{\longmapsto} \textsc{decl}(\mu) \text{ as } \{m_j : \overline{U} \to \textsc{int}\}$$

There are multiple possible assignments for constraint variables μ and $\overline{\varepsilon}$, and for code completion to work, the solver must find all of them. The following table shows the possible assignments for μ, $\overline{\varepsilon}$, T, $\overline{U}$, and $\overline{V}$ that the solver might yield:

Solution	μ	$\overline{\varepsilon}$	T	$\overline{U}$	$\overline{V}$
Solution 1	$\mathtt{a}_2$	[ ]	INT	[ ]	[ ]
Solution 2	$\mathtt{b}_3$	$[\varepsilon_1, \varepsilon_2]$	INT	[INT, INT]	$[\tau_1, \tau_2]$
~~Solution 3~~	$\mathtt{c}_8$	[ ]	BOOL	[ ]	[ ]
Solution 4	$\mathtt{d}_9$	$[\varepsilon_1]$	INT	[INT]	$[\tau_1]$

Note that solution 3 is not valid, as it tries to assign $T \mapsto$ BOOL whereas T had previously already been assigned INT. Also note how the solver could infer lists of constraint variables for $\overline{U}$ and $\overline{V}$. But, as before, we would not want the solver to keep expanding on the constraint variables it has introduced. If we had not relaxed these variables such that they may remain free, the solver would have to find some assignment for the variables that satisfies them. In this example, the solver might have added a method call to an arbitrary method with a compatible return type, such as $\mathtt{a}_2$. In other scenarios, the solver may not be able to find such a solution, or find infinitely many.

The solutions returned by the solver can be turned into syntax fragments and presented to the user as code completion proposals, where we replace the free constraint variables by syntax placeholders. The order of the proposals is not determined by the solver, as we consider this to be a separate concern. For example, we may want to order the proposals by their frequency of use, or use the semantic model to order the proposals by closeness (e.g.,

local variables before global variables). In this example, code completion would propose the following method calls:

```
a2()
b3($Exp, $Exp)
d9($Exp)
```

Summary To use the semantics of the programming language for code completion, we first need a semantic specification that includes a model for name binding. This is already provided by the scope graphs used by the Statix constraint solver. However, the solver also needs to support returning incomplete solutions. The solver needs to be able to distinguish between constraint variables that we want to have solved and those that may remain free, and we need to be able to indicate how deep we want a given constraint variable to be solved. By using the semantic rules, a solution can include syntactic assignments to variables. Finally, the solver must be able to return more than one solution, so we can display all of them to the user as part of code completion.

2.6 Extract Definition

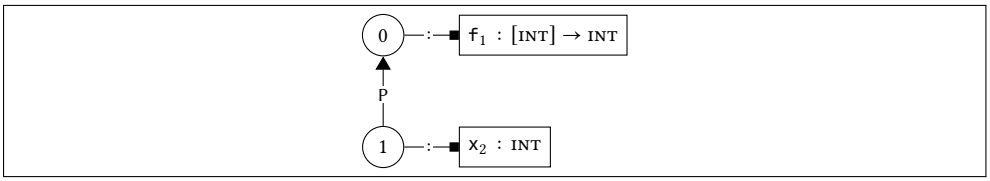
A common refactoring is the *extract definition* refactoring, where the user selects a subexpression, and the refactoring replaces any occurrences of that expression with a reference to a variable definition initialized by the subexpression. In the example in Fig. 2.4, we want to extract the `x - 3` subexpression into a separate definition. We assume the program is syntactically complete and semantically correct.

The first step in this refactoring is to determine the new syntax that we expect as a result of the refactoring. This is language-specific syntax, selected by the user and specified in advance by the language developer. The syntax fragment uses placeholders, as shown below, where *$Type* is a placeholder for the type of the newly created variable and *$ID* is a placeholder for a variable name. In this case, we want all three occurrences *$ID* to refer to the same variable:

```
int f1(int x2) {
    ($Type) ($ID) = x3 - 3;
    return ($ID) + ($ID) * x5;
}
```

We create a copy of the previous, valid, solution returned by the solver, and adapt it to this refactoring. This is a two-part process: relaxing the solution, and adding new constraints to the problem. Relaxing the solution removes any variables, resolutions, constraints, and scope graph nodes that are no longer valid or that impact the aspects we want

```
int f1(int x2) {
    return (x3 - 3) + (x4 - 3) * x5;
}
```



$$\text{(J-MethodDec)}\ \frac{\begin{array}{c} s \vdash [\![C_i]\!] \Rightarrow T \quad \nabla s_m \quad s_m \xrightarrow{P} s \\ s \vdash [\![\overline{C_k}]\!] \Rightarrow \overline{T_k} \quad s_m \xrightarrow{:} \overline{p_k} : \overline{T_k} \\ s_m \vdash \overline{b}\ \text{OK} \quad s \xrightarrow{:} m_j : \overline{T_k} \to T \\ s_m \vdash e : U \qquad \vdash U <: T \end{array}}{s \vdash C_i\ m_j(\overline{C_k}\ \overline{p_k})\ \{\ \overline{b}\ \texttt{return}\ e;\}\ \text{OK}}$$

$$\text{(J-VarDec)}\ \frac{s \vdash [\![C_x]\!] \Rightarrow T \quad s \xrightarrow{:} v_y : T \quad s \vdash e : T}{s \vdash C_x\ v_y = e;\ \text{OK}}$$

$$\text{(J-Var)}\ \frac{\text{query}\ s \xrightarrow{P^* S^* :} \text{DECL}(x_i)\ \text{as}\ x_j : T}{s \vdash x_i : T}$$

$$\text{(J-IntBinOp)}\ \frac{s \vdash e_1 : T_1 \quad s \vdash e_2 : T_2 \quad T_1 = T_2 = T = \text{INT}}{s \vdash e_1 \oplus e_2 : T}$$

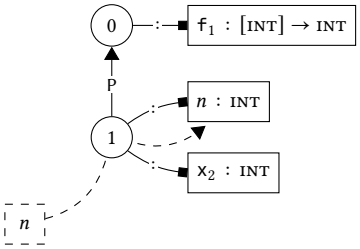
Figure 2.4: Java program before applying the *extract definition* refactoring, and the corresponding scope graph and relevant Statix typing rules.

to refactor. For extracting a definition, relaxation involves only removing the reference relation $x_4 \mapsto x_2$, since the reference x_4 has been removed. However, we still want the variable references x_3 and x_5 to resolve to the same definition x_2.

Now we can add new constraints to the problem, but the refactoring should add only those constraints that result from the changed syntax. The constraints may contain syntax terms, but we replace any occurrences of the placeholders with constraint variables. For the type placeholder *$Type*, we will use the constraint variable τ. Since we want all occurrences of the *$ID* placeholder to refer to the same variable, we should replace all occurrences with the same constraint variable.

Where other approaches use the solver to find a concrete name for the variable (Steimann, 2018), we argue that this is not necessary for the solver to give a correct result. We merely want to indicate to the solver that the new name is different from all other names in the

```
int f(int x) {
    int n = x - 3;
    return n + n * x;
}
```



	n	τ	T	v_j
Solution	n	`int`	INT	n

Figure 2.5: Java program after applying the *extract definition* refactoring, and the corresponding scope graph and variable assignments. Note that n in the program is a rigid variable, which has yet to be assigned a concrete name.

program. This gives a separation of concerns: the solver can verify that the program satisfies the constraints without needing to produce any concrete names, and generating the concrete names can be done externally after the solver has verified the program. For example, some IDEs provide a list of name suggestions that they generate from the context, such as the type of the expression.

To distinguish the abstract name from any other name in the program, we can use a rigid variable: one that is distinct from any other variable or name. Similar to how rigid variables are used to create new distinct scopes in the scope graph (through ∇s), we create a new rigid variable to represent the name of the newly introduced variable: ∇n. Due to rules (J-VarDec) and (J-Var), this results in the following constraints:

$$\nabla n$$

$$1 \vdash [\![\tau]\!] \Rightarrow T \qquad\qquad 1 \xrightarrow{:} n : T$$

$$1 \vdash (\mathsf{x}_3 - 3) : T \qquad\qquad \text{query } 1 \xmapsto{\mathsf{P}^*\mathsf{S}^*:} \textsc{decl}(n) \text{ as } x_j : T$$

Solving these constraints results in the variable assignment and scope graph shown in Fig. 2.5.

From this, we can conclude that the refactoring is valid, does not semantically change the program (since the existing constraints and reference resolutions are preserved), and that the type of the newly introduced variable is `int`. However, to finish the refactoring, we

have to decide on a concrete name for rigid variable n.

A concrete name can be provided by the refactoring tool or by the user. In any case, we can test whether the suggested name is allowed by reinvoking the solver with the new solution and one additional constraint: to constrain n to the chosen name, say `i`.

$$n = \texttt{i}$$

The new constraint may result in an invalid solution, for example, when the chosen name overlaps with another, or causes inadvertent name capture somewhere in the program. In this example, `x` is not allowed as a concrete name for n.

However, if this results in a valid solution, the concrete name is acceptable and the refactoring can finish. In this example, it would produce the following code:

```
int f(int x) {
  int i = x - 3;
  return i + i * x;
}
```

Summary As part of the refactoring, we generate new syntax, where we use placeholders to indicate where we need more information. In the newly generated constraints, we have a constraint variable taking the place of every placeholder, which allows us to use the solver to find a solution to the problem. By using a rigid variable in place of a concrete name, we can indicate that the name is different from all other names in the program without having to specify such a name concretely, giving a separation of concerns between finding whether the program is valid, and what concrete name to choose.

2.7 Related Work

Erdweg, van der Storm, Völter, Boersma, et al. (2013) identify editor services as an important aspect of language workbenches, and give an overview of commonly supported editor services. However, Omar, Voysey, Hilton, Sunshine, et al. (2017) have pointed out that the study of the semantic foundations of editors and editor interactions has received little attention so far. We discuss work related to code completion and refactoring, as those are most relevant to the editor services we covered in this chapter.

Reference Resolution Language workbenches, such as Xtext (Eysholdt and Behrens, 2010) and Spoofax (Kats and Visser, 2010b; Visser, Wachsmuth, et al., 2014), provide support for language-parametric reference resolution based on declarative name-binding specifications. Xtext supports specification of references in the language grammar as crosslinks,

which specify the sort that an identifier can refer to. Xtext will check the validity of the references and add them to the model.

The first approach to declarative name-binding specification in Spoofax was the NaBL name-binding language (Konat et al., 2013). The name-binding rules defined the definition sites and their scopes based on the abstract syntax of the program. The built-in reference resolution algorithm could only create an index in which references can be looked up, which limits its flexibility to be used in other editor services.

Instead, the approach we use in this chapter uses the expressiveness of the constraints and the flexibility of the Statix constraint solver to enable reference resolution to be used in various editor services.

Code Completion The Xtext and Spoofax language workbenches also provide support for language-parametric syntactic completion, based on a syntax definition. In the case of Xtext, it suggests possible keywords. Spoofax suggests complete syntactic constructs, and represents incomplete syntax trees using placeholders that act like holes in the program text (de Souza Amorim, Erdweg, et al., 2016). This representation is instrumental for translating an incomplete program to an abstract syntax tree with variables, which allows us to use it in a constraint context. A program with placeholders is similar to the representation of an AST with holes that is common in structure editors. Although structure editors are primarily concerned with guaranteeing that the program is well-typed with respect to the abstract syntax signature, recent work investigates editors that also maintain other well-formedness properties, such as well-typedness.

The Hazelnut editor provides a language-parametric structure editor that guarantees well-typed ASTs for languages whose type system is defined in a bidirectional style (Omar, Voysey, Hilton, Aldrich, et al., 2017). JastAdd extends its reference attribute grammars to provide a context-sensitive completion service that suggests the names of variables and functions, but still requires some language-specific effort to derive these suggestions (Söderberg and Hedin, 2011). Steimann, Frenkel, and Voelter (2017) use constraint-based language specifications to ensure edits preserve well-formedness. They focus on an architecture that allows interaction between the solver and the user during the editing process, to resolve conflicts that may have been introduced. We aim to generate semantic completion proposals by combining the mechanisms for syntactic completion with checking and inference based on the language specification. Another important difference is that issues around name resolution are largely ignored in their work, because references are actual references in the underlying model, whereas in our text-based setting, we need to consider naming issues.

Semantic code completion also has similarities to interactive proof search, such as offered by proof assistants. For example, the editor of Agda (Coquand, Takeyama, and Synek,

2006) features holes that are similar to placeholders. An automatic procedure tries to find proof terms (expressions) that fit the goal (type). There are some important differences with our approach to code completion. The procedure to find these terms is not language-parametric, but specific to Agda. The search procedure does not exploit the typing rules, but duplicates knowledge from the type checker. Type correctness is guaranteed by type-checking the fragment after it is generated.

Refactoring There is a long line of research on the specification and implementation of refactorings. Tip et al. (2011) study type-related refactorings, such as adding type parameters, extracting interfaces, and pulling up methods. They use type constraints to specify the invariants that ensure correct behavior. Steimann (2018) and others extend this work to include constraints for other aspects, such as access modifiers and names. By representing the program itself using constraint variables, both the invariants and the refactoring intent can be represented as constraints. Finding the refactored program, within the limits of the given constraints, is delegated to the constraint solver. This approach is in many aspects similar to ours, and hopefully the techniques they developed for performance carry over to our approach. An important difference is that by using Statix, the constraint solver is aware of the complete binding model. In their approach, preventing capture requires the introduction of inequality constraints between names. These constraints do not follow from regular constraint-based typing rules. In our approach, the resolution constraints that are part of the typing rules can also be used to ensure the invariance of name resolution during refactoring.

2.8 Conclusion

In this chapter, we have discussed various semantic editor services, and shown how they can be expressed in terms of the semantic rules, constraints, and scope graph. We show that Statix constraints are expressive enough to formulate interesting editor services. We have pointed out that the Statix solver used for type checking is not suitable for the scenarios that arise in editor services, and we have identified several requirements that such alternative solver strategies should have. The main requirements we identified are:

- The solver must be able to try different alternatives, guess values, backtrack on failed attempts, and able to return multiple solutions. As we have shown, this is a requirement to implementing code completion, but also for other editor services such as find usages and find all subclasses.

- Instead of always searching for complete solutions (i.e., assignments to all variables), the search should be controlled by user-defined criteria, including whether the solver should only consider deterministic inference, or whether it tries to find solutions non-deterministically. These criteria should be able to depend on variables appearing in the constraints. Specifying which constraint variables may remain unconstrained, and how deep the solver should search for an assignment, allows us to direct the search to finding solutions to only those constraint variables we are interested in, and prevent the solver from getting stuck.

Finally, we propose to extend the mechanisms of creating scopes in Statix to a general mechanism of rigid variables. These rigid variables can be used to solve problems around inventing new concrete names in the constraint solver, and should make it easier to implement various refactorings without having to deal with concrete names, accidental variable capture, and ambiguous names. Factoring out the choice of finding concrete names separates concerns, and also allows for language-specific strategies (e.g., suggesting variables names based on types).

Future Work This chapter presented ideas on what would be needed for language-parametric semantic editor services. To verify our approach, we need to implement the proposed extensions to the Statix solver. This will allows us to evaluate their feasibility in practice, as it is not clear whether implementing some of these techniques, such as having the solver backtrack and trying to find multiple possible solutions, would cause performance issues or introduce non-termination. And if so, how we could avoid that without impacting the expressiveness of the semantic rules too much.

There are editor services, other than those we discussed, to which we could apply our approach, such as *fix import*, *search for symbol*, and in particular *rename refactoring*. Rename refactoring is interesting because it not only needs to rename the references to the renamed declaration, but possibly other references and declarations as well. For example, when a method is renamed, all overriding methods need to be renamed too, and this relation is not visible in the program model, but only encoded in the semantic rules.

While our current approach is focused on preserving the static semantics of the program, for certain refactorings it may be required to extend the approach to also preserve certain dynamic aspects of the semantics. Additionally, a combination of our approaches might be used to implement program generation that is guaranteed to produce programs that are semantically correct.

3

Language-Parametric Static-Semantic Code Completion

Many semantic editor services can be specified as constraint problems, as envisioned in the previous chapter. In this chapter, we explore and implement a code completion editor service parameterized by the static-semantic specification of the language. This chapter is a slightly revised version of the paper previously published as:

> Daniel A. A. Pelsmaeker, Hendrik van Antwerpen, Casper Bach, and Eelco Visser (2022b). "Language-Parametric Static Semantic Code Completion." In: *Proceedings of the ACM on Programming Languages* 6.OOPSLA1, pp. 1–30. DOI: 10.1145/3527329

Abstract

Code completion is an editor service in IDEs that proposes code fragments for the user to insert at the caret position in their code. Code completion should be sound and complete. It should be sound, such that it only proposes fragments that do not violate the syntactic and static-semantic rules of the language. It should be complete, such that it proposes all valid fragments so that code completion can be used to construct all programs. To realize soundness and completeness, code completion should be informed by the language definition. In practice, the implementation of code completion is an additional effort in the implementation of a language.

In this chapter, we develop a framework for language-parametric semantic code completion for statically typed programming languages based on their specification of syntax and static semantics, realizing the implementation of a code completion editor service with minimal additional effort. The framework builds on the SDF3 syntax definition formalism and the Statix static-semantic specification language. The algorithm reinterprets the static-semantic definition to find sound expansions of predicates and solutions to name-resolution queries in scope graphs. This allows a search strategy to explore the solution space and synthesize completion proposals. The implementation of the strategy language and code completion algorithm extend the implementation of the Statix solver, and can be used for any language defined in Statix. We demonstrate soundness and completeness of the completion proposal synthesis, and evaluate its performance.

3.1 Introduction

An *Integrated Development Environment (IDE)* assists programmers to produce high-quality code by means of *editor services* such as syntax highlighting, code completion, code navigation, inline error messages, and refactorings. *Code completion* is an editor service that allows the user to insert a code fragment at the current caret position by selecting it from a list of proposals provided by the editor. This reduces typing, minimizes typing errors, helps the user find relevant code fragments, aids in discovery by proposing possible syntax and references, and avoids incorrect API usage.

Ideally, code completion should be *sound*. That is, it should take into account the syntactic and static-semantic rules of the programming language, to only propose syntax and references that do not introduce additional compile-time errors at the caret position. Because code completion that proposes nothing is trivially sound, it should ideally also be

complete. That is, it should propose all permissible syntax and references given the compile-time model of the program.

In practice, implementations of code completion are generally complete in terms of proposed names, but vary in soundness. That is, they may propose names and syntax that are not correct in the context of the caret position according to the syntactic and static-semantic rules of the program. For example, proposing names whose (return) type does not match the expected type at the caret position, or proposing references that, when inserted, would inadvertently capture another name than intended. We discuss more examples below.

Many code editors provide code completion out of the box. But to support a wide range of programming languages, code completion is often either too generic or too specific and not generalizable. The simplest implementations of code completion, such as used in Notepad++, propose all the words that are present in the current document or project, without any regard for their semantics or the context at the caret. This will propose keywords and identifiers equally, regardless of whether the keywords are actually correct, whether declarations are actually reachable at the caret position, and whether the proposed identifier inadvertently captures another name. While this is language-agnostic, and thus easy to implement, it is also the least useful.

More advanced code completion implementations propose only references to declarations that are reachable and visible from the caret position, according to the static semantic rules of the language, and these are the most common form of code completion in IDEs. For example, VS Code, Visual Studio, Eclipse, and IntelliJ IDEA use this kind of code completion by default for the languages they fully support. While this does not prevent the user from inserting a reference of the wrong type, it does allow the user to discover the available names. Additionally, inserting a reference of the wrong type is often useful when the reference is used as the receiver of a chain of method calls and field accesses.

Syntax-aware code completion, such as proposed by de Souza Amorim, Erdweg, et al. (2016), propose only syntax fragments that can be inserted at the caret position according to the syntactic rules of the language. However, this approach does not take the semantics of the fragment into account, allowing, for example, a Boolean and-expression to be inserted as the initializer for an integer variable. Furthermore, syntactic code completion does not take the binding context into account and does not propose references to actual variables and functions.

Type-directed development in languages such as Agda (Norell, 2007) and Idris (Brady, 2013) uses programs with explicit holes to represent incompleteness. Completion commands can involve proof search to determine valid proposals. Unfortunately, such proposals are not always guaranteed to be sound. Ideally, code completion in a static context proposes

only syntax and references that are valid according to both the syntactic and static semantic rules of the language. Selecting any proposal provided by this code completion will not introduce any additional syntactic or static semantic errors in the program.

Code completion is often reimplemented for each language and for each editor supporting that language. Such implementations may require significant development effort, and vary in features, soundness, and completeness. Instead, we would like to derive a sound and complete code completion editor service from the formal definition of a language automatically, to reduce development effort and increase reliability. Language workbenches (Fowler, 2005) aim to reduce the effort required to implement a programming environment for a language. Language workbenches provide significant support for developing parsers and type checkers from language definitions. However, the implementation of code completion in language workbenches such as Xtext (Xtext Team, 2021), MPS (Jetbrains, 2021), Rascal (Klint, van der Storm, and Vinju, 2009; Klint, van der Storm, and Vinju, 2010; Klint, van der Storm, and Vinju, 2019), and Spoofax (Kats and Visser, 2010b) either require additional work on top of a language definition, are unsound, and/or incomplete. (We discuss the state-of-the-art of code completion in language workbenches extensively in Section 3.7.) In Chapter 2, we speculated on the derivation of a range of editor services from the declarative, constraint-based specification of the static semantics of a language. We built upon this proposal in order to automatically derive code completion from a language definition.

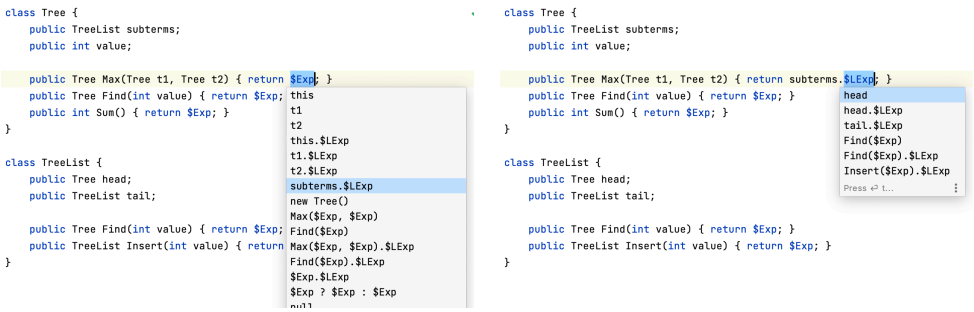
In this chapter, we develop a framework for language-parametric semantic code completion for statically typed programming languages based on their syntax definition and static-semantic specification, realizing the implementation of a code completion editor service with minimal additional effort. The framework builds on the SDF3 syntax definition formalism (de Souza Amorim and Visser, 2020) and the Statix static-semantic specification language (van Antwerpen, Néron, et al., 2016; Rouvoet et al., 2020) and is integrated into the Spoofax language workbench (Kats and Visser, 2010b). Our code completion algorithm reinterprets the declarative static-semantic definition of a language in Statix. We use its order-independent constraints and syntax-directed predicates to drive a search strategy that uses the scope graph and Statix solver to explore the space of possible code completion solutions in a languag-parametric way. Our Statix reinterpretation includes the implementation of name-resolution queries with non-ground subjects. The implementation of the strategy language and code completion algorithm abstracts and extends the implementation of the Statix solver, and can be used for any language defined in Statix. We evaluate the performance, soundness, and completeness of the completion proposal synthesis.

In summary, our contributions are:

- a language-parametric code completion editor service that is derived automatically from the static semantics of a language;
- code completion proposals that are sound and complete by construction;
- a definition of code completion as a language-parametric constraint resolution problem on incomplete programs with holes/placeholders using syntax-directed semantic rules in Statix, see Section 3.4;
- extensions to the Statix solver interface to expand predicate constraints into their syntax-directed alternatives and to expand query constraints into their possible resolutions;
- a set of strategy combinators to compose these primitives, and the composition of a search strategy for sound and complete code completion, as presented in Section 3.5;
- the integration of our language-parametric semantic code completion in the IntelliJ IDEA and Eclipse editors generated by the Spoofax language workbench; and
- the evaluation of the algorithm by applying it to language definitions for Tiger (Appel, 2002) and ChocoPy (Padhye, Sen, and Hilfinger, 2019) in Section 3.6. Using test suites of 135 Tiger and 136 ChocoPy programs (60 624 test cases of up to 1 091 lines of code) we determined soundness and completeness, and measured performance. The median performance of code completions is 422 ms, but has outliers exceeding 4 seconds for certain large test files with many declarations in scope.

We submitted the source code and benchmark files related to the original publication as an artifact (Pelsmaeker, van Antwerpen, Bach, et al., 2022a).

Outline We proceed as follows. In the next section, we analyze the requirements for sound and complete code completion. In Section 3.3 we discuss the prerequisites for our approach. In Section 3.2 we show the code completion algorithm on a running example, and its implementation using search strategies is detailed in Section 3.5. In Section 3.6 we evaluate the soundness, completeness, and performance of our implementation. We discuss related work in Section 3.7 and future work in Section 3.8. We conclude in Section 3.9.



(a) Completion on a placeholder of type `Tree`.

(b) Completion on the inserted proposal.

Figure 3.1: Code completion in action: the user invokes code completion on the selected placeholder and gets presented a complete list of completion proposals that are syntactically and semantically sound.

3.2 Code Completion

In this section, we discuss the requirements for sound and complete semantic code completion. Throughout the chapter, we use Micro-C#, a small subset of the C# language, as a running example object language. Figure 3.1a shows how code completion displays a list of proposals when the user performs code completion on the selected syntactic placeholder `$Exp`. The list includes both syntactic proposals, such as the ternary operator `$Exp ? $Exp : $Exp`, as well as references, such as to variable `t1` and method call `Max($Exp, $Exp)`.

Representing Incompleteness Code completion is often, but not necessarily, invoked on an incomplete program, where one of the proposals is used to get one step closer to a complete program. For the purpose of this chapter, we assume that the program has a syntactic placeholder near the caret position to represent a hole in an incomplete program. In the Spoofax language workbench, a placeholder is typically named after the syntactic sort of the missing phrase. For example, `$Exp` is a placeholder representing an expression, which ensures that incomplete programs are syntactically correct. Placeholders can be inserted using placeholder inference (de Souza Amorim, Erdweg, et al., 2016), which can infer placeholders for both complete and incomplete programs, or as part of the parsing process of a syntactically incomplete program.

Syntactic Code Completion As a first step, code completion should propose only syntactically correct fragments. De Souza Amorim, Erdweg, et al. (2016) describe the synthesis of a complete list of syntactically correct proposals following the grammar of the language, by proposing phrases corresponding to the productions of the syntactic sort of the placeholder. Incomplete sub-phrases are represented by new placeholders. For example, a syntactically valid proposal for `$Exp` is `$Exp + $Exp`.

Filtering on Types When only considering the syntactic sort of the placeholder, not all syntactically correct proposals will produce fragments that are also type-correct. For example, completing an expression in a context where an integer-typed expression is expected will also propose expressions that produce Boolean-typed expressions. Therefore, code completion should only propose syntactically correct fragments that are also type-correct in the completion context. Depending on the language, there might be additional static semantic constraints on proposals, such as subtype and supertype constraints, or shape constraints on vectors in dependently typed systems. The proposals should adhere to such constraints.

Completing Names Just proposing the syntax of an unspecified reference or function call (e.g., `$ID` or `$ID($Args)`) is often not very useful. Ideally, code completion should propose the actual names of variables and calls to functions that are available in the scope of the completion context. We could even improve upon this by proposing function calls with placeholders for the correct number of arguments, or record initialization syntax with initializers for each of the fields. This requires computing the names that are available at the completion context, taking into account name binding rules such as scoping, shadowing, importing, and overriding.

Soundness and Completeness Finally, code completion should be sound and complete. Soundness ensures that inserting the proposal does not introduce additional syntactic or static semantic errors. This is achieved by taking into account the guidelines above only to propose phrases that are syntactically and semantically correct at the point of insertion with respect to the static semantics of the language. Completeness ensures that *all* syntactic constructs and reachable identifiers (modulo shadowing) that are permitted according to the syntax and static semantics of the language are proposed, up to one level deep, modulo literals and the identifiers of declarations. With a complete code completion algorithm, one can (re)create any valid program through repeated application of code completion, but repeated application of code completion does not necessarily result in a complete program. In this chapter, we determine whether a program is valid by checking it against the static-semantic specification of the language. Therefore, our notion of completeness is limited to what is possible to express and check with the static-semantic rules.

3.3 Background: Language Definition in SDF3 and Statix

Our code completion editor service is derived from the formal definition of the syntax and static semantics of a language. We build on the SDF3 syntax definition formalism (de Souza Amorim and Visser, 2020) and Statix static-semantic specification language (van Antwerpen, Bach, et al., 2018b; Rouvoet et al., 2020) and their implementations in the Spoofax language workbench (Kats and Visser, 2010b), but note that SDF3 and Statix are not contributions of this chapter. Figures 3.2 to 3.4 show excerpts of the SDF3 and Statix specifications of our Micro-C# language (the full specifications are in Appendix A.1 and Appendix A.2, respectively). The rest of this section provides a high-level introduction to SDF3 and Statix to illustrate key features for this chapter. We refer to the cited literature for more details.

3.3.1 Syntax Definition in SDF3

SDF3 is a language for the formal definition of the syntax of (domain-specific) programming languages (de Souza Amorim and Visser, 2020). In addition to a basic syntactic description based on context-free grammars, the language provides several features to aid the definition of language-aware editors. The syntactic rules relevant for this subsection are in Fig. 3.2.

Concrete and Abstract Syntax The syntax of language constructs is defined through context-free productions of the form `A.C = `α with `A` a non-terminal, `C` a constructor symbol, and α a list of symbols. The constructor symbol creates a direct correspondence between parse tree and abstract syntax tree. For example, the expression `1 + x` has abstract syntax tree `Add(Int("1"), Var("x"))` according to the productions in Fig. 3.2. Keywords and layout (whitespace and comments) are omitted from the abstract syntax.

Template Productions SDF3 extends plain context-free grammar productions with *template* productions to support pretty-printing abstract syntax trees to text. A template quotes a text template with anti-quotation for non-terminal symbols. For example, the template `MemAcc.Call = <<ID>(<{Exp ", "}*>)>` specifies that no spaces should be used around the parentheses of the call, but that spaces *should* be inserted after the commas between arguments. This entails that code completion can generate an abstract syntax tree as a completion proposal, which can then be translated automatically to a pretty-printed concrete syntax fragment.

Placeholders SDF3 already provides language-parametric support for *syntactic* code completion by generating an editor service that synthesizes completion proposals based on a syntax definition (de Souza Amorim, Erdweg, et al., 2016). Incompleteness in a program is represented explicitly through placeholders, syntax which is added automatically to the

syntax definition. That is, for each syntactic sort `S`, a production `S.S-Plhdr = "$S"` is automatically generated. For example, `2 + $Exp` represents an expression with a hole. Code completion is a transformation on the abstract syntax tree, transforming placeholders to AST representations of completion proposals. For example, an *addition* proposal is synthesized by rewriting `Exp-Plhdr()` to `Add(Exp-Plhdr(), Exp-Plhdr())`. Textual completions are produced by pretty-printing the AST proposals.

Disambiguation Instead of encoding the disambiguation rules of a language in the productions of the syntax definition, they are encoded in separate associativity and priority declarations. For example, the sentence `a * b + c` is parsed as `Add(Mul(Var("a"), Var("b")), Var("c"))` since multiplication has higher priority than addition, as specified in Fig. 3.2. These declarative disambiguation rules are not just used for parsing, but also during pretty-printing to ensure that brackets are inserted where necessary to preserve the abstract syntax tree structure. For example, when the placeholder in `a * $Exp` is expanded into an addition, the result is pretty-printed as `a * ($Exp + $Exp)`.

Syntax API In summary, SDF3 provides a representation and API for syntax definition, parsing, and pretty-printing, including support for the representation of incomplete programs and syntactic completion proposals.

3.3.2 Static Semantics in Statix

Statix is a declarative language for describing the static semantics of programming languages in terms of equality and scope graph constraints (van Antwerpen, Bach, et al., 2018b). The Statix implementation automatically derives a type checker from a specification. The implementation is based on an operational semantics that is sound with respect to the declarative semantics (Rouvoet et al., 2020). In the rest of this chapter, we will show how we automatically derive a code completion editor service from a Statix specification. In this section, we give a brief overview of the relevant features of Statix. Excerpts from the Statix specification of Micro-C# in Appendix A.2 are shown in Figs. 3.3 and 3.4.

Equality Constraints Statix is a constraint-based specification language. The basic constraints are equality constraints `t1 == t2` and inequality constraints `t1 != t2` on terms. An equality constraint is satisfiable if its term arguments are equal or can be unified. An inequality constraint is satisfiable if its term arguments are not equal. The quantification of existentials `{T}` enables specification of type inference. The constraints are order-independent, which we use in our code completion algorithm to be able to explore different alternatives in parallel by expanding upon unsolved constraints.

```
context-free sorts
  Exp LValue LExp MemAcc

context-free syntax
  Exp.New        = <new <Type>(<>)>
  Exp.Int        = <<INT>>
  Exp.Mul        = <<Exp> * <Exp>>     {left}
  Exp.Add        = <<Exp> + <Exp>>     {left}
  Exp.Cond       = <<Exp> ? <Exp> : <Exp>>
  Exp            = <(<Exp>)>         {bracket}
  Exp            = LValue
  LValue.Member  = <<Exp>.<LExp>>     {right}
  LValue         = LExp
  LExp.Project   = <<MemAcc>.<LExp>> {right}
  LExp           = MemAcc
  MemAcc.Var     = <<ID>>
  MemAcc.Call    = <<ID>(<{Exp ", "}*>)>

context-free priorities
  { Exp.New LExp.Project
    MemAcc.Var MemAcc.Call
  } > Exp.Mul > Exp.Add > Exp.Cond,

  LValue.Member <0> .> Exp = LValue,

  LValue.Member <0> .> { Exp.New Exp.Mul
    Exp.Add Exp.Cond }
```

Figure 3.2: Excerpt of Micro-C# SDF3 syntax definition.

```
declareVar : scope * ID * TYPE
declareVar(s, decl, T) :-
  !type[Var{decl@decl}, T] in s.

typeOfVar : scope * ID -> TYPE
typeOfVar(s, name) = T :- {decl}
  resolveVar(s, name) == [(_,(decl,T))|_].

resolveVar : scope * ID -> list(DECL)
resolveVar(s, name) = ps :-
  query type
    filter P*
    and { d' :- d' == Var{name@_} }
    min $ < P
    in s |-> ps.
```

Figure 3.3: Excerpt of Micro-C# Statix name resolution.

```
typeOfExp(s, New(t)) = T :- {cs}
  typeOfType(s, t) == T@CLASS(_, cs).

typeOfExp(s, Add(e1, e2)) = INT() :-
  subtypeOf(typeOfExp(s, e1), INT()),
  subtypeOf(typeOfExp(s, e2), INT()).

typeOfExp(s, Cond(ec, e1, e2)) = T :-
  {T1 T2}
  subtypeOf(typeOfExp(s, ec), BOOL()),
  typeOfExp(s, e1) == T1,
  typeOfExp(s, e2) == T2,
  lub(T1, T2) == T.

typeOfExp(s, LValue2Exp(lvalue)) =
  typeOfLValue(s, lvalue).

typeOfLValue(s,
  LExp2LValue(MemAcc2LExp(Var(x))))
  = typeOfVarOrMember(s, x).

typeOfLValue(s, LExp2LValue(le)) =
  typeOfLExp(s, s, le).

typeOfLValue(s, Member(e, le)) = T :- {cs}
  typeOfExp(s, e) == CLASS(_,cs),
  typeOfLExp(s, cs, le) == T.

typeOfLExp(s, cs, Project(ma, le)) = T :-
  {T1 cs2}
  typeOfMemAcc(s, cs, ma) == CLASS(_,cs2),
  typeOfLExp(s, cs2, le) == T.

typeOfLExp(s, cs, MemAcc2LExp(ma)) =
  typeOfMemAcc(s, cs, ma).

typeOfMemAcc(s, cs, Var(name)) = T :-
  typeOfMember(cs, name) == T.

typeOfMemAcc(s, cs, Call(x, es)) = TR :-
  {TA}
  typeOfMember(cs, x) == METHOD(TA, TR),
  typesOfExps(s, es) == TA.

typeOfExp(_, Exp-Plhdr()) = BOTTOM().
```

Figure 3.4: Excerpt of Micro-C# Statix typing rules.

Specifying Name Binding with Scope Graphs Scope graph constraints (van Antwerpen, Néron, et al., 2016) are a key feature of Statix. A scope graph is an abstraction of a program to its name-binding facts. A scope is a vertex in the graph that corresponds to a region of a program that behaves uniformly with respect to name binding. Declarations of names are data in scopes. Edges between scopes encode reachability. A declaration in a scope is reachable from all scopes with paths (sequences of edges) to that scope.

The language-agnostic representation allows high-level reasoning about name resolution (Neron et al., 2015). A named reference is bound to its corresponding declaration through a name-resolution query constraint. A name-resolution query for a reference starts in a scope to find a path along edges to a matching declaration. References are ambiguous if multiple matching declarations can be reached. Disambiguation can be expressed through path well-formedness and specificity ordering on paths. For example, the query in Fig. 3.3 defines lexical shadowing of variables by finding paths along zero or more `P` edges (`P*`), preferring shorter paths to longer paths (`$ < P`), where `P` edges in the scope graph represent the lexical parent relation between scopes. For code completion we use the name resolution and scope graph of Statix to find all reachable declarations for query constraints where the subject is not ground.

Predicates Predicate symbols are used to introduce user-defined constraints that abstract from basic unification and scope graph constraints. In general, a predicate defines a relation on terms and scopes. Rules define whether a predicate holds for a combination of terms. A rule has the form $p(t_1, \ldots, t_n)$ `:-` $c_1, \ldots, c_n$ and defines a predicate p through the conjunction of a number of constraints $c_1, \ldots, c_n$. A predicate can be defined by means of multiple rules, as long as the heads of the rules are mutually exclusive (including specificity, see below). (Or in operational terms, rule selection in Statix is non-backtracking.) Additionally, rule selection is 'syntax directed', which we use in our algorithm can propose syntax fragments.

A functional predicate defines a one-to-one relationship between input terms and output term. For example, the predicate `typeOfExp(s, e) = T` associates a single type `T` with the combination of a scope `s` and an expression `e`. From the perspective of specifying static semantics, a predicate is interpreted as an assertion about the well-boundness and well-typedness of abstract syntax terms. For example, this rule defines that a method call has type `TR` if the method has type `METHOD(TA, TR)` and the arguments have types `TA`:

```
typeOfMemAcc(s, cs, Call(x, es)) = TR :- {TA}
  typeOfMember(cs, x) == METHOD(TA, TR),
  typesOfExps(s, es) == TA.
```

Rules with overlapping patterns are ordered by means of a specificity ordering. That is, a rule with a more specific pattern in its head has priority over (and excludes) rules with

more general patterns. Furthermore, the binding of patterns in rule heads is done through pattern matching rather than unification, thus the (input) arguments of predicates needs to be ground (enough).

From Declarative to Operational Semantics Statix has a declarative semantics that defines when a solution satisfies a set of constraints. A language engineer should only have to consider the declarative semantics. The Statix solver implements the operational semantics of Statix, which schedules the solution of query constraints such that their results are guaranteed to be stable, i.e., provide the same results as they would in the final scope graph (Rouvoet et al., 2020).

In Section 3.5, we extend the interface of the solver in order to give alternative interpretations to Statix specifications. In particular, we use the solver to get all possible alternatives of a predicate and all possible resolutions of a query, and use these in the code completion algorithm to explore the solution space and synthesize proposals. Finally, we use the solver to reject unsound proposals.

3.4 Constraint-Based Code Completion

Given an incomplete program, the goal of code completion is to synthesize a list of completion proposal phrases that can be substituted for the placeholder being completed, such that the resulting program is well-bound and well-typed. That is, the program after substituting a completion proposal for a placeholder should not violate any constraints when type-checked (that were not violated before invoking code completion). The key idea of our approach is to use the constraint-based definition of the type checker to synthesize the completion proposals, in order to guarantee that the proposals are well-bound and well-typed by construction.

The core of our language-parametric code completion algorithm is a search strategy that explores the solution space through the static-semantic specification of the object language. In this section, we explore that search strategy by tracing the three steps of the synthesis of the list of completion proposals for the example program in Fig. 3.1a: *predicate expansion*, *injection expansion*, and *query expansion*, using the rules from Figs. 3.3 and 3.4. In the next section, we give a formal definition of the algorithm.

3.4.1 Preliminary Analysis

The preliminary step in the code completion algorithm is the analysis of the incomplete program, in which we replace the placeholder to be completed (the first `$Exp` in Fig. 3.1a) with a constraint variable. We will refer to this constraint variable as the *completion variable*.

Synthesizing completion proposals entails finding all terms that can be substituted for the completion variable without violating any constraints.

Constraint solving in Statix first solves basic constraints and simplifies predicates via rule application. Statix only simplifies predicates if the input arguments are ground enough. That is, the arguments are at least as ground as the patterns in the heads of the predicate rules. Constraints that are not ground enough cannot be solved, which is typically the case for constraints involving the completion variable. For the example program in Fig. 3.1a, after the preliminary analysis we have the residual constraint


```
typeOfExp(s, e) == CLASS("Tree", cs)
```

with `e` being the completion variable corresponding to placeholder `$Exp`, `s` the scope of the completion variable, and `cs` the scope of the class definition. These unsolved constraints are the starting point for the code completion algorithm.

3.4.2 Predicate Expansion

In the first step of the code completion algorithm, the completion variable is expanded by unifying the residual constraint with the heads of predicate rules, leading to its instantiations. The syntax-directed nature of the rules ensures that all terms that are type-correct will be considered. Each rule leads to a possible alternative. We will examine several representative rules.

The rule for the *conditional* expression is defined as follows:

```
typeOfExp(s, Cond(ec, e1, e2)) = T :- {T1 T2}
  typeOfExp(s, e1) == T1,    typeOfExp(s, e2) == T2,
  subtypeOf(typeOfExp(s, ec), BOOL()),  lub(T1, T2) == T.
```

The above rule for the *conditional* expression declares that a *conditional* expression with condition `ec`, then-expression `e1` and else-expression `e2` is well-typed when the type of the condition is a subtype of Boolean, and the types of the branches `e1` and `e2` share a least-upper-bound type `T`, which has to equal the type of the *conditional* expression as a whole. Unifying the residual constraint to the head of this rule leads to the constraints:

```
e == Cond(ec, e1, e2),  CLASS("Tree", cs) == T,  subtypeOf(typeOfExp(s, ec), BOOL()),
typeOfExp(s, e1) == T1, typeOfExp(s, e2) == T2,  lub(T1, T2) == T.
```

Since the unification constraint `CLASS("Tree", cs) == T` is trivially satisfiable, and the other constraints are non-ground, this leads to the completion proposal `Cond(ec, e1, e2)`, or `$Exp ? $Exp : $Exp` in concrete syntax. Note that the predicate expansion step is *only applied once* to a non-ground constraint. Recursive application of this strategy to the residual, non-ground constraints synthesized by this step could lead to infinitely many proposals, even

though each expansion only introduces a finite number of expansions. However, not all rules lead to a completion proposal. For example, consider the rule for *addition*:

```
typeOfExp(s, Add(e1, e2)) = INT() :-
  subtypeOf(typeOfExp(s, e1), INT()),  subtypeOf(typeOfExp(s, e2), INT()).
```

Unifying the residual constraint to the head of this rule leads to the constraints

```
e == Add(e1, e2),                    INT() == CLASS("Tree", cs),
subtypeOf(typeOfExp(s, e1), INT()),  subtypeOf(typeOfExp(s, e2), INT()).
```


Since the unification constraint `INT() == CLASS("Tree", cs)` is not satisfiable, this alternative is discarded and does not lead to a completion proposal. And it should not, since `Add(e1, e2)` would lead to an ill-typed term when substituted for the placeholder. Finally, consider the rule for *new*:

```
typeOfExp(s, New(t)) = T :- {cs}
  typeOfType(s, t) == T@CLASS(_, cs).
```

Unifying the residual constraint to the head of this rule leads to the constraints

```
e == New(t),   T == CLASS("Tree", cs),
typeOfType(s, t) == T@CLASS(_, cs).
```

Since the unification with `T` is satisfiable, this leads to a completion proposal `New(t)` (or `new($Type)` in concrete syntax), requiring the type `t` to be `Tree`. Following this procedure also for the rule `typeOfExp(s, LValue2Exp(lvalue))`, we arrive at the following proposals:

```
$Exp ? $Exp : $Exp,    new $Type(),    $LValue
```

This list only includes well-typed (syntactic) completion proposals, but includes another placeholder `$LValue`. We will improve this in the next step.

3.4.3 Injection Expansion

An injection is a grammar production of the form `S1 = S2`, that includes all `S2` constructs in `S1`. For example, in the Micro-C# syntax definition, we have defined the `LValue` sort to be used as expressions and as the left-hand side of assignments. The inclusion of `LValue` in `Exp` is an injection since no additional syntax is added. In fact, there is a chain of injections `Exp = LValue = LExp = MemAcc` to include variables and calls as expressions:

```
Exp.Add       = <<Exp> + <Exp>>    {left}
Exp.Cond      = <<Exp> ? <Exp> : <Exp>>
Exp           = <(<Exp>)>        {bracket}
Exp           = LValue
LValue.Member = <<Exp>.<LExp>>    {right}
```

```
LValue        = LExp
LExp.Project  = <<MemAcc>.<LExp>> {right}
```

In Statix, injections get an explicit constructor. For example, the `Exp = LValue` production becomes `Exp.LValue2Exp = LValue` with typing rule

```
typeOfExp(s, LValue2Exp(lvalue)) == typeOfLValue(s, lvalue).
```

Thus, the expansion procedure above expands the completion variable to

```
e == LValue2Exp(lvalue), typeOfLValue(s, lvalue).
```

leading to the completion proposal `LValue2Exp(lvalue)` (`$LValue` in concrete syntax); not a very useful proposal.

Recognizing the special status of injections, the second step of the algorithm expands injections, which includes the constructs of the sub-sort in the set of proposals. That is, recognizing `LValue2Exp(lvalue)` as an injection marks `lvalue` as a completion variable to be considered for expansion, following the procedure of predicate expansion above. Considering the full set of typing rules for LValues in Fig. 3.4, code completion now replaces the `$LValue` proposal with `Member(e, le)`, `Project(m, le)`, `Var(x)`, and `Call(x, es)` (or `$Exp.$LExp`, `$MemAcc.$LExp`, `$ID`, and `$ID($Exps)` in concrete syntax), leading to the following list of proposals:

```
$ID,            $ID($Exps),            $Exp.$LExp,
$Exp ? $Exp : $Exp,   new $Type(),   $MemAcc.$LExp
```

The list contains only well-typed syntactic completion proposals. This includes placeholders for referencing variables and calling methods, but not yet the actual identifiers of those that are in scope. We will address this next.

3.4.4 Query Expansion

So far, we have been concerned with synthesizing completion proposals that are syntactically correct and type correct. However, to get truly useful proposals, we also want to propose those identifiers of the right type that are in scope. As the proposals above have constraint variables instead of concrete names, this requires performing name-resolution queries with non-ground subjects.

To understand the issues involved, we examine the case of variables. One rule for `typeOfLValue` uses the `typeOfVarOrMember(s, name)` predicate to assert that the name refers to

a valid local variable or class member. For the sake of this presentation, we simplify that to just asserting that the name refers to a valid local variable, defined by the following rules:

```
typeOfExp(s, LValue2Exp(lvalue)) =
  typeOfLValue(s, lvalue).
typeOfLValue(s, LExp2LValue(MemAcc2LExp(Var(x)))) =
  typeOfVar(s, x).
typeOfVar(s, name) = T :-
  resolveVar(s, name) == [(_, (_, T))].
resolveVar(s, name) = ps :-
  query type filter P* and { d' :- d' == Var{name@_} } min $ < P in s |-> ps.
```

The predicate `typeOfVar` is just an abstraction for a pattern match on the result of the name-resolution query defined by `resolveVar`. But it does illustrate that after injection expansion, regular constraint resolution by the Statix solver unfolds the predicate `typeOfExp(s, e)` into the constraints

```
e == LExp2LValue(MemAcc2LExp(Var(x))), T == CLASS("Tree", cs), ps == [(_, (_, T))],
query type filter P* and { d' :- d' == Var{x@_} } min $ < P in s |-> ps.
```

In other words, we are looking for a concrete name for the `x` constraint variable, such that the query results in a path to a single declaration with the type `CLASS("Tree", cs)`.

The query expansion step of the algorithm interprets a non-ground query and produces all possible alternatives. That is, it finds all names of declarations such that executing the query with that name makes the query succeed. We describe the technical details of query expansion in Section 3.5. For the example program of Fig. 3.1a, expanding the query finds the arguments `t1` and `t2` of type `CLASS("Tree", _)` and the instance variable `subterms` of type `CLASS("TreeList", _)`. Since we are looking for an expression of type `Tree`, only `t1` and `t2` end up in the list of proposals. Similarly, we find completion proposals for `Calls` to methods `Max` and `Find`, since they are in scope and return `Tree` objects.

```
t1,     t2,     $ID,     Max($Exps),     Find($Exps),     $ID($Exps)
$Exp ? $Exp : $Exp,     new $Type(),     $Exp.$LExp,     $MemAcc.$LExp,
```

3.4.5 Rule Specialization

Consider again the syntax of method calls and variables in Section 3.4.3. As we saw above, we do get proposals for variables and method calls, but by default, the algorithm will propose quite shallow syntax, only one level deep. However, to create a chain of dot expressions, we need to first complete `$Exp` to `$MemAcc.$LExp` and then complete the `$MemAcc` placeholder to a call. This would have to be done by the user, as the code completion algorithm only synthesizes one level of expansion in order to avoid infinite recursion.

While this approach is sound and complete, in cases like this, we would like code completion to also expand nested constraint variables, whereas in other cases the current behavior is fine, such as in `$Exp + $Exp`. Resolution queries are a possible indication that it is desirable to further expand a constraint. However, queries may be hidden behind a series of predicate abstractions as we saw in the case of variables above. We have not succeeded in identifying a reliable heuristic to selectively expand constraints in these particular cases.

Fortunately, we have identified *rule specialization* as a simple approach for the language engineer to direct the code completion algorithm to provide more concrete and useful proposals for these particular cases, without code duplication and without affecting the generality of the definition or the soundness of the approach. For example, consider the following rules for member access projection (dot expressions):

```
// Rules for method calls
typeOfLExp(s, cs, lv@Project(_, _))           = typeOfProjectExp(s, cs, lv).
typeOfLExp(s, cs, lv@Project(Var(_), _))      = typeOfProjectExp(s, cs, lv).
typeOfLExp(s, cs, lv@Project(MCall(_, _), _)) = typeOfProjectExp(s, cs, lv).

typeOfProjectExp(s, cs, Project(ma, le)) = T :- {T1 cs2}
  typeOfMemAcc(s, cs, ma) == CLASS(_, cs2), typeOfLExp(s, cs2, le) == T.
```

Here, we have captured the rule for dot expressions in a new predicate `typeOfProjectExp`. We define the `typeOfLExp` predicate for the general case of a dot expression, and we extend it with a rule for the special cases `Project(Var(_), _)` and `Project(MCall(_, _), _)`. This instructs the predicate expansion of Section 3.4.2 to also synthesize completion proposals for these cases. Applying such rule specializations to `New` as well, leads to a list of more specific completion proposals in addition to the generic syntactic proposals:

```
t1,     t2,     $ID,     Max($Exps),     Find($Exps),     $ID($Exps)
t1.$LExp,     t2.$LExp,     subterms.$LExp,     this,     this.$Exp,
new Tree(),       new $Type(),       $MemAcc.$LExp,      $Exp.$LExp,
Max($Exps).$LExp,       Find($Exps).$LExp,       $Exp ? $Exp : $Exp.
```

Thus, by a slight refactoring of the specification, we can guide the code completion algorithm to produce more useful completion proposals. A rule specialization adds only a single additional rule to be expanded by predicate expansion, and therefore does not incur more performance overhead than the other rules in the specification.

```
rules // placeholders
  programOk(Module-Plhdr()).
  declOk(_, Decl-Plhdr()).
  typeOfExp(_, Exp-Plhdr()) = _.
```

Figure 3.5: Example static-semantic rules for accepting placeholders.

3.4.6 Filtering and Ranking

3

The completion proposals often include some placeholders for unspecified literals (e.g., string literals, integers, and identifiers), such as the `$ID` placeholder for an identifier. In general, it is impractical to expand on these placeholders and list all possible literal values, but for completeness and to aid in language discovery, we decided not to filter these placeholders from the list of proposals.

Ranking the completion proposals aims to offer the most likely choices to the user first. However, as ranking the proposals is out of the scope of this chapter, we chose to simply rank them by specificity, putting the proposal with the least number of placeholders first. Thus, we present to the user the final list of proposals for the completion of `$Exp` in example program of Fig. 3.1a:

```
 t1,       t2,       this,       new Tree(),       subterms.$LExp,
t1.$LExp,    t2.$LExp,    this.$Exp,    Find($Exps),    Max($Exps),
 $ID,     new $Type(),     Find($Exps).$LExp,     Max($Exps).$LExp,
$ID($Exps),    $MemAcc.$LExp,    $Exp.$LExp,    $Exp ? $Exp : $Exp,
```

3.4.7 Applying Code Completion

A primary motivation of this work is that enabling language-parametric semantic code completion for a Spoofax language takes very little additional effort. Any Spoofax language which has its static semantics defined using Statix requires only adding rules that permit placeholders and changing rules that catch all remaining constraints, to enable sound and complete static semantic code completion.

Rules for Placeholders Syntactic placeholders in Spoofax are just constructor application terms of the form `Exp-Plhdr()`. As placeholders are inserted explicitly when choosing a completion proposal, the language developer has to add some extra rules to the Statix specification to allow it to accept these placeholder terms, as shown in Fig. 3.5. In the future, these rules could be generated automatically based on the syntactic and semantic specifications.

Catch-All Rules Some language developers write *catch-all* Statix rules that, for example, generate a warning about unsupported syntax. As Statix rules are tried in order of specificity, such catch-all rules are tried last and therefore work as expected during the normal

code analysis. However, such rules consume the remaining constraints that apply to the placeholder being completed. This prevents code completion from finding completions, as there will be no constraints left for the code completion algorithm to expand upon and this impacts completeness.

```
typeOfExp(_, _) = _ :- try { false } | warning $[Not implemented].
```

Instead, the language developer should be more specific about the syntax. For example:

```
typeOfExp(_, Plus(_, _)) = _ :- try { false } | warning $[Not implemented].
```


3.5 Code Completion Algorithm

In the previous section, we illustrated the main steps of the code completion algorithm. In this section, we formally define the algorithm as a composition of basic solver functions using a strategy language to search the solution space. In the actual implementation, we use Java, the implementation language of the Spoofax language workbench (Kats and Visser, 2010b), to implement the strategies. However, we use Haskell notation to present the algorithm in this chapter because Java is quite verbose.

3.5.1 Constraint Solver

The code completion algorithm builds on the Statix constraint solver, which performs syntax-directed constraint solving and unification based on the operational semantics of Statix (Rouvoet et al., 2020). The solver takes a declarative specification and a solver state (shown in Fig. 3.6), which consists of a set of unsolved constraints, a unifier, a scope graph, and a set of error messages, and some other data that is only used by the Statix solver internally.

When analyzing a file during normal error-checking operation, the solver gets an initial solver state with only the program's root predicate constraint (e.g., `programOk()`) applied to the AST of the program, represented as a *term*. For example:

```
programOk(Program([ Class("TreeList", ...), Class("Tree", ...) ])) .
```

The solver expands predicate constraints that are sufficiently ground into their constituent constraints, and then performs unification to remove as many solved (or unsolvable) equality constraints from the solver state as possible. Equality constraints whose variables the solver cannot unify contribute errors to the set of errors. The remaining unsolved constraints, those that are not ground enough, remain in the resulting solver state.

Code completion is performed on an incomplete program, where syntactic placeholders represent (typed) holes in the program. These holes are represented by constraint variables in the solver state, which inevitably results in some unsolved constraints due to not being

ground enough. The core idea of the code completion algorithm is to use the solver to try different expansions of the solver state's unsolved constraints to explore the space of possible solutions for a particular *completion variable*, the constraint variable that corresponds to the syntactic placeholder being completed.


```
data SolverState = SolverState {
  constraints :: Set Constraint
, unifier :: Unifier
, scopeGraph :: ScopeGraph
, messages :: Set Message
, spec :: Spec
} deriving Eq;
```

Figure 3.6: Solver state structure.

3.5.2 Search Strategies

To explore the space of possible code completion solutions, we employ a language-parametric *search strategy* that is itself a composition of other *strategies*. A strategy is a function of type `a → [b]`, which defines a computation that takes an input and optional arguments, and produces a list of alternatives. If the list is empty, the strategy is said to have failed. In particular, many of these strategies are *search strategies* of type `SolverState → [SolverState]`, which take a solver state and produce a (possibly empty) list of solver states. We compose the strategies using *strategy combinators*, which are basic primitive strategies that take other strategies as parameters.

Search strategies give us a functional way to compose more basic strategies, and the flexibility to experiment with different strategy compositions to find the most optimal implementation. Additionally, we expect that other editor services that use the static analysis of a program could be implemented language-parametrically using different compositions of strategies.

3.5.3 Built-in Search Strategies

The code completion algorithm uses the existing Statix solver, persistent solver state structure, name resolution algorithm, and the static-semantic specification of the object language. Additionally, the algorithm requires access to the pretty-printer of the object language, which is automatically generated from the language's syntax specification.

Our contribution is an abstraction of these details of the Statix solver and Spoofax internals into a handful of search strategies, shown in Fig. 3.7, providing a well-defined interface for their interaction with the code completion algorithm.

```
type State = SolverState
type Constr = Constraint

selectPredicate     :: ((Constr, State) → Bool) → State → [(Constr, State)]
selectQuery         :: ((Constr, State) → Bool) → State → [(Constr, State)]
expandPredicate     :: (Constr, State) → [State]
expandQuery         :: (Constr, State) → [State]
infer               :: State → [State]
assertNoErrors      :: Set Message → State → [State]

containsVar         :: [Var] → Constr → Bool
findInjection       :: Var → State → [(Var, State)]
projectVar          :: Var → State → [Term]
prettyPrint         :: Term → [String]
```

Figure 3.7: Built-in search strategies.

Selecting Constraints. The `selectPredicate` and `selectQuery` strategies inspect the solver state, select a number of predicate or query constraints, respectively, and return these as constraint-state pairs. The function that is passed as the first argument determines whether a constraint is selected, such as the `containsVar` strategy which is used to check whether the constraint contains the completion variable. These constraint-state pairs are the input to one of the built-in `expandPredicate` and `expandQuery` strategies.

Expanding a Predicate The `expandPredicate` strategy takes a constraint-state pair and determines all possible rules that the predicate constraint could expand to. For each rule, the strategy produces a new solver state with the rule's constraints added to it. In Statix, as rules are tried in order of specificity, a more general rule for a predicate implies that all more specific rules were already tried and rejected. Because our algorithm has to produce order-independent expansions, we make these rejections of more specific rules explicit in the expansion of each rule. Section 3.4.2 gives examples of the application of this strategy. The `expandPredicate` strategy itself does not verify whether the resulting solver states are sound, this is done in a separate step.

Expanding a Query The `expandQuery` strategy expands the selected query constraint into all possible sets of declarations the query could resolve to, while taking shadowing into account. This strategy calls the name resolution algorithm that is also used by normal Statix inference. Normally, Statix only performs name resolution when the query arguments are sufficiently ground—that is, the scope and name of the query are known—but as we are trying to find all declarations, we do not have a name to resolve. We first query for all reachable names without applying the query's shadowing rules, to avoid applying shadowing across *all* names in the program. This gives us all names reachable from the current scope, for each of which we invoke the name resolution again but this time we apply the

normal name and shadowing rules to get all reachable declarations with correct shadowing applied. Most query constraints match the result to a singleton set, asserting that exactly one declaration is expected, but it is entirely possible in Statix for a query to match to a larger set. Therefore, this strategy constructs all *sets* of reachable declarations while taking shadowing into account. See Section 3.4.4 for an example.

Inference The `infer` strategy takes a solver state with unsolved constraints and invokes the Statix constraint solver to perform inference and solve as many constraints as possible. The strategy does not fail if the resulting solver state contains error messages. We deal with erroneous solver states explicitly in a separate step, which we will detail next.

Asserting Well-Typedness The `assertNoErrors` strategy inspects a solver state and returns a singleton list of this state if it contains no errors messages other than those that were present when code completion was invoked (which are passed as a set to the strategy). However, if the solver state contains new errors, the strategy returns an empty list, indicating failure. This ensures that code completion will not generate proposals that are unsound with respect to the static semantics of the language.

Finding an Injection The projection of the completion variable in the solver state might contain constraint variables for injections, such as `lv` in `LValue2Exp(lv)`. The `findInjection` strategy finds all these constraint variables and returns a list of tuples of these variables and the input solver state. This allows these variables to be expanded by another step in the algorithm.

Projecting a Variable The syntax-directed semantic rules use *terms* to represent fragments of the AST of a program. The solver state does not directly contain the values of variables but instead has a unifier that associates a partial term with each variable, where holes in the term are represented by other variables that may or may not be present in the unifier. The `projectVar` strategy projects the term for a given variable out of the solver state's unifier, by finding the partial term for a variable, and recursively repeating this for each variable in the term until only un-unified variables remain.

Pretty-Printing a Term The completion proposals should be represented to the user using the syntax of the object language. *Pretty-printing* converts a term into its corresponding syntax using the syntax rules of the language. In the Spoofax language workbench, this is achieved by calling the `pp` pretty-printing Stratego term-rewriting strategy that is generated for each language from its syntax definition. The `prettyPrint` search strategy takes a term and invokes the `pp` term-rewriting strategy to get its string representation. Any constraint variables that were left in the term are pretty-printed as syntactic placeholders in the object language. For example, the term `Add(e, Int("3"))` would be pretty-printed as `$Exp + 3`.

```
id     :: a → [a]  -- Identity
seq    :: (a → [b]) → (b → [c]) → a → [c] -- Sequential composition
or     :: (a → [b]) → (a → [b]) → a → [b] -- Disjunction
try    :: (a → [a]) → a → [a]  -- Apply strategy if possible
repeat :: (a → [a]) → a → [a]  -- Repeat strategy until failure
fixSet :: Eq a ⇒ (a → [a]) → a → [a]  -- Repeat strategy until nothing changes
limit  :: Int → (a → [b]) → a → [b]  -- Limit sequence resulting from strategy
(.>)    = seq
```

Figure 3.8: Strategy combinators.

3.5.4 Combining Strategies

The built-in strategies above provided core operations on solver states. We built the code completion algorithm by combining these built-in strategies using strategy combinators inspired by the Stratego transformation language (Visser, Benaissa, and Tolmach, 1998). Figure 3.8 defines the type signatures of the strategy combinators. We include the definitions of the combinators in Appendix A.5.

3.5.5 Code Completion Algorithm

Our code completion algorithm composes the built-in strategies using the strategy combinators, to perform a directed search of the space of all possible completion proposals for a given completion variable. The list of proposals must be sound and complete. That is, it must include all reachable declarations (modulo shadowing) and all syntactic constructs up to one level deep (modulo literals and declaration names), as defined in the static-semantic specification of the object language. We present the algorithm in Fig. 3.9.

The core of the code completion algorithm is the `complete` strategy, which performs three steps to gather all valid proposals: expand predicates, expand injections, and expand queries. These correspond to the descriptions in Sections 3.4.2 to 3.4.4.

Expanding Predicates The first step is to `expandAllPredicates`, which takes a solver state, and selects exactly one predicate constraint that contains the completion variable. It removes this constraint and expands it into its possible alternatives through the built-in `expandPredicate` strategy. Unsound solver states are rejected through `infer` and `assertNoErrors`. This process is repeated until all predicate constraints that contain the completion variable have been expanded.

Note that this only expands the completion variable for one level, as the newly added constraints in the expanded solver states describe the expansion of the completion variable, and therefore no longer contain the completion variable itself. This is desirable, since it prevents the algorithm from infinitely recursing to expand, say, `$Exp` into `$Exp + $Exp` into `($Exp + $Exp) + $Exp` and so forth.

```
-- Core completion algorithm: find the completion proposals
complete :: Var → Set Message → SolverState → [SolverState]
complete v ms
  = (expandAllPredicates v ms) .>
    (expandAllInjections v ms) .>
    (expandAllQueries    v ms)

-- Expand predicate constraints that contain the completion variable
expandAllPredicates :: Var → Set Message → SolverState → [SolverState]
expandAllPredicates v ms =
  repeat (
    (limit 1 (selectPredicate (\(c, s) → containsVar [v] c))) .>
    (expandPredicate) .>
    (infer) .>
    (assertNoErrors ms)
  )

-- Expand predicate constraints on injections that contain the completion variable
expandAllInjections :: Var → Set Message → SolverState → [SolverState]
expandAllInjections v ms
  = fixSet (try (
      (findInjection v) .>
      (\(vInj, s) → (complete vInj ms s))
    ))

-- Expand all query constraints that contain the specified variable
expandAllQueries :: Var → Set Message → SolverState → [SolverState]
expandAllQueries v ms
  = or id (
    repeat (
      (limit 1 (selectQuery (\(c, s) → (containsVar [v] c)))) .>
      (expandQuery) .>
      (infer) .>
      (assertNoErrors ms)
    ))

-- Computes the completion proposals and filters, orders, and pretty-prints them
getCompletionProposals :: Var → Set Message → SolverState → [String]
getCompletionProposals v ms
  = (complete v ms) .>
    (projectVar v) .>
    prettyPrint
```

Figure 3.9: The code completion strategy.

Expanding Injections As the previous step only expands predicate constraints for one level deep, in some solver states the completion variable will have been expanded into the explicit constructor `B2A` of an injection `A = B`. The `expandAllInjections` strategy expands these injections by applying `findInjection` to find the states in which the completion variable has been bound to an injection constructor, and then invokes `complete` recursively to expand the injections into their alternatives. This is repeated, using `fixSet`, until either the strategy fails because there are no injection constructors to expand, or the set of solver states no longer changes. Since `complete` only returns sound solver states, there is no need to check whether the states are sound.

If injections are (indirectly) recursive, the algorithm could end up in an infinite loop. To avoid this, we track the names of the injections that have been expanded so far during this step and avoid expanding them again.

Expanding Queries Any unsolved query constraints that remain in the solver state will not be ground enough for the normal solver. The `expandAllQueries` strategy selects and expands one of the query constraints that contains the completion variable using `expandQuery`. Ill-typed states are again rejected by applying `infer` and `assertNoErrors`, and this is repeated until there are no more query constraints to expand. The `or` strategy is used to include the original input state in the resulting set of solver states, such that the original state with the placeholder is retained regardless of whether are query constraints.

Presenting to the User The algorithm starts and finishes at the `getCompletionProposals` strategy, which takes the completion variable, set of pre-existing error messages, and the solver state constructed from the initial analysis of the program. This is a solver state with some unsolved constraints, due to the completion variable. Once the strategy gathers the completion proposals through calling `complete`, it projects the terms of the completion variables out of the unifier and pretty-prints them in the syntax of the object language. These are the completion proposals that will be shown to the user.

3.5.6 Strategy Implementation

We implemented all strategies in Java, abstracted boilerplate code for strategies into a runtime, and integrated code completion in the Spoofax language workbench and the Eclipse and IntelliJ IDEA editors. Spoofax with its meta-languages is about 2 million lines of Java code, including almost 400 000 lines of code for the Statix language and implementation. Our code completion algorithm uses the existing Statix structures, and calls its solver and name resolution algorithm with no changes except some bug fixes. Table 3.1 shows the number of lines of Java code (excluding comments and blank lines) we contributed to enable language-parametric static semantic code completion in Spoofax.

Table 3.1: Code completion lines of Java code.

Strategies Runtime	5 921 loc
Built-in Strategies (3.5.3)	1 333 loc
Strategy Combinators (3.5.4)	1 816 loc
Code Completion (3.5.5)	401 loc
Spoofax Integration	2 068 loc
Eclipse UI Integration	133 loc
IntelliJ UI Integration	100 loc
Total	11 772 loc

To implement search strategies efficiently, there were some considerations to address.

Persistent Solver State While exploring the solution space we create many slightly different instances of the solver state, its unifier, scope graph, and other data. This can have a significant impact on the performance of the algorithm due to copying lots of data, and associated garbage collector pressure. To mitigate this, we use the efficient persistent (immutable) data structures of Steindorfer (2017), which are also used by Statix internally. This means that it is cheap to fork a solver state to explore alternatives.

Lazy Lists While developing our algorithm, we observed cases in which the strategies performed much more work than necessary due to strict evaluation. For example, if the strategy `foo` returns a hundred solver results, then the strategy `(limit 1 foo)` would return only the first and discard the other 99. To avoid this extra work, we implemented the strategies such that they are not immediately evaluated to return a list, but instead return a pull-based *lazy list*, whose next value is only computed when requested.

3.6 Evaluation

While we used the smaller Micro-C# language as an illustration language for this chapter, we chose to use the Tiger and ChocoPy programming languages for our evaluation, as they include some more interesting type system and name binding constructs and already have extensive test suites.

Tiger is a small Algol-based programming language described in Appel (2002) used to demonstrate both functional and object-oriented compiler concepts, and is used as a standard example language for testing the Spoofax language workbench. Its specification in the Spoofax language workbench consists of 114 syntactic productions and 32 Statix predicates defined with 75 rules in 513 lines of Statix code (excluding generated files for the constructor declarations), covering the entire language. We include the full specification of Tiger in Appendices A.3 and A.4.

The ChocoPy language (Padhye, Sen, and Hilfinger, 2019) is a statically-typed subset of Python used to teach programming languages and compiler construction courses. Among other features, it demonstrates inheritance, method overloading, and nested functions. It is specified in Spoofax using 125 syntactic productions and 49 Statix predicates defines with 165 rules in 698 lines of Statix code (again, excluding generated files for the constructor declarations).

Tiger and ChocoPy have extensive test suites consisting of 166 Tiger files[1] and 182 ChocoPy files[2], that test various aspects of the languages. We selected all test files that were not testing syntactic or semantic errors, and ended up with a test suite of 114 Tiger and 129 ChocoPy files. We synthetically enlarged 21 of the Tiger files and 7 of the ChocoPy files by adding and copying additional declarations, and included these in the test sets. We used these 135 Tiger and 136 ChocoPy files to evaluate the code completion algorithm in three ways:

1. are the proposed completions correct according to the syntax and static semantics of the language (soundness);
2. is it always possible to select a completion that would reconstruct (the start of) the elided subtree (completeness); and
3. how long does it take for code completion to provide the proposals (performance).

[1] https://github.com/MetaBorgCube/metaborg-tiger/
[2] https://github.com/cs164berkeley

3.6.1 Soundness

The proposals generated by the code completion algorithm are sound by construction with respect to the static-semantic specification of the object language, as they call upon the Statix constraint solver to verify whether a proposal is acceptable. Unsound proposals (i.e., those that introduce errors that were not initially present in the program) would be found by the Statix solver (through the `infer` strategy), removed from the results (through the `assertNoErrors` strategy), and not be shown to the user.

3.6.2 Completeness

Our notion of completeness of the algorithm implies that it will propose all syntactic constructs up to one level deep for which a static semantic rule has been defined, and all identifiers that are reachable from the current scope in the scope graph (modulo shadowing), provided that these proposals are sound. Repeated invocation of code completion allows users to construct any semantically valid program, modulo declaration names and literals.

To show with high confidence the completeness of the code completion algorithm, we constructed 23 424 Tiger and 37 200 ChocoPy test cases from the selected 135 Tiger files and 136 ChocoPy files. Each test case is a variant of a full program, but with one abstract syntax subtree replaced by a placeholder. This mimics the most common case where the user invokes completion on the only placeholder (or inferred placeholder) of the program. The test asserts that invoking code completion on this placeholder results in a set of completion proposals from which one can reconstruct some of the elided code, showing that we can use repeated invocation of code completion to reconstruct each of the test files.

All tests succeeded, showing both that the static-semantic specification of Tiger and ChocoPy covers all the syntax in the test files, and that the algorithm always produced a proposal matching the expected syntax or reference. In other words, the language-parametric code completion algorithm is complete with respect to the static-semantic specification.

3.6.3 Performance

To measure the performance of the code completion algorithm, we use the same 60 624 constructed test cases, where we measure the time from invoking code completion on the placeholder until it produces its proposals. To avoid local user processes interfering with the tests, we ran them on an otherwise unused server with two 32-core AMD Epyc 7452 processors and 256 GB of memory, but note that both our algorithm and the Statix solver only support single-threaded operation and do not utilize the large amount of memory. We used Java 17 and limited the process to 4 GB of memory and 4 processors to align with the system that a typical user might have available.

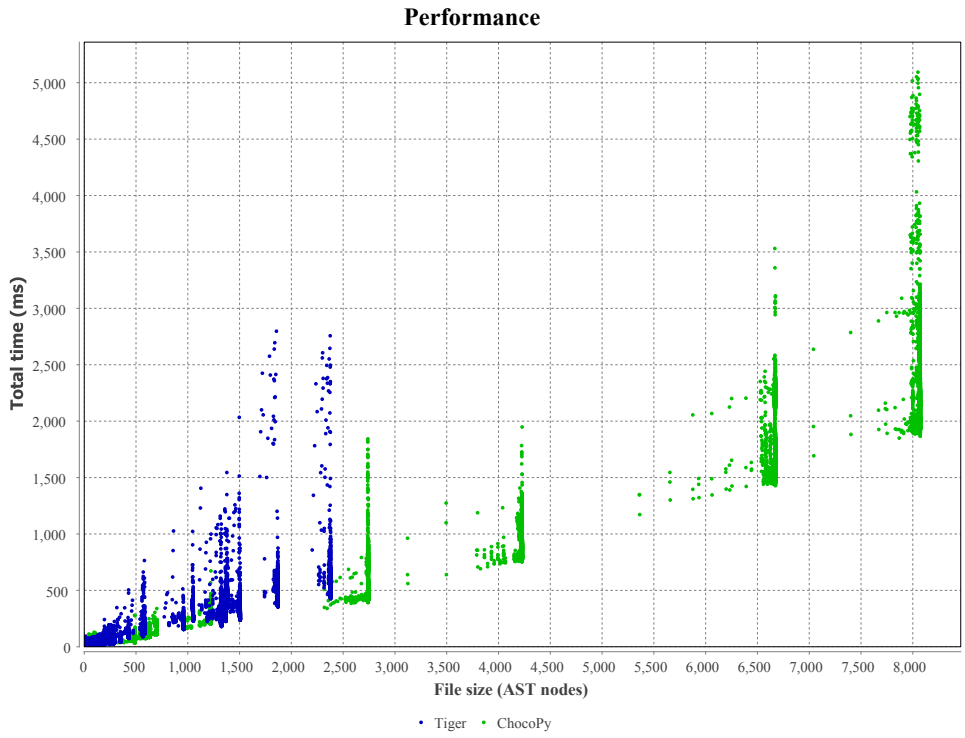


Figure 3.10: Code completion algorithm performance as a scatter plot, showing the time spent on each of the 60 624 Tiger and ChocoPy code completion invocations against the number of AST nodes in the test file.

In Fig. 3.10 we show a scatter plot of the time, in milliseconds, that each test case took from being invoked until presenting the list of proposals. Tiger tests (up to 2 368 AST nodes) are in blue, whereas ChocoPy tests (up to 8 060 nodes) are in green. The tests are horizontally plotted by the number of nodes in the abstract syntax tree in which completion is being invoked, and vertically by the time they took, in milliseconds.

The performance results are outlined in Table 3.2, split into three performance aspects: the time for the initial analysis of the program with the placeholder inserted (*Initial Analysis*), the time for the code completion search algorithm (*Code Completion*), and the total time that includes both and some minor housekeeping. The total time reflects and affects the user experience, as it is the time from code completion being invoked until it presents a list of proposals.

Table 3.2: Benchmark results.

	Average	Median	95th pctl	99th pctl	Maximum
Tiger (Code Completion)	61 ms	35 ms	180 ms	414 ms	2 719 ms
Tiger (Initial Analysis)	206 ms	206 ms	458 ms	486 ms	836 ms
Tiger (Total)	267 ms	259 ms	595 ms	837 ms	2 808 ms
ChocoPy (Code Completion)	144 ms	18 ms	710 ms	991 ms	3 158 ms
ChocoPy (Initial Analysis)	967 ms	815 ms	2 029 ms	2 077 ms	2 871 ms
ChocoPy (Total)	1 111 ms	919 ms	2 603 ms	2 889 ms	5 104 ms
All (Code Completion)	113 ms	32 ms	675 ms	894 ms	3 158 ms
All (Initial Analysis)	683 ms	371 ms	1 991 ms	2 058 ms	2 871 ms
All (Total)	797 ms	422 ms	2 280 ms	2 785 ms	5 104 ms

3.6.4 Analysis

The test results indicate that 70.60% of code completion invocations in our test set can be completed in 1 000 ms or less, from being invoked to presenting the code completion proposals. In 99.17% of cases the code completion algorithm itself completes within 1 000 ms, and the rest is accounted for by the initial analysis. In 114 test cases (0.19%) the tests ran significantly slower, taking more than 4 seconds to conclude for certain large test files with many declarations in scope.

The performance of this initial analysis depends on both the complexity of the static-semantic specification and the size of the program, and accounts for the empty space at the bottom of the graph in Fig. 3.10. As the test programs for ChocoPy get larger, the initial analysis also takes more time. However, developers are working on making the analysis in Statix faster (and eventually incremental), and this will also bring the total time down.

The performance of the code completion search strategy itself is governed by the number of reachable declarations and the performance of the `expandQuery` strategy, which searches the scope graph for all possible (sets of) declarations a query could resolve to. In the worst case, for n declarations this will try to find all 2^n power sets of declarations, but in practice, most declarations only resolve to a singleton set.

Distinct Checks A pattern commonly written by language developers in their Statix specification is to check whether a declaration is *distinct* by adding the declaration to the scope graph, followed by an assertion that it is the only declaration in the scope graph:

```
declareType(s, x, T) :-
  s -> Type{x} with typeOfDecl T,                // add declaration to scope graph
  typeOfDecl of Type{x} in s |-> [_].            // assert declaration is distinct
```

Performing code completion on the identifier placeholder of a declaration will expand on the query constraint to find existing declarations. However, applying any of these decla-

rations will always fail, as that would add a second declaration to the scope graph, making the query no longer satisfiable. Therefore, code completion rejects all existing declarations, which is sound, but this fruitless search of the scope graph does have a measurable negative performance impact. In practice, this will not be a problem, as users rarely invoke code completion on the name of a declaration.

Permissive Lookups Similarly, another pattern sometimes used is to allow *permissive lookups* when performing a query. This ensures that a possible error about duplicate declarations is shown on the declaration site rather than the reference site.

```
lookupType(s, x) = T :- {x'}
  typeOfDecl of Type{x} in s |-> [(_, (Type{x'}, T)), _].
```

However, this query will cause our code completion algorithm to try all possible sets of declarations that the query could resolve to, even though only the singleton sets are relevant. Again, this has no impact on the soundness of the algorithm but does have a performance impact.

3.6.5 Threats to Validity

The performance tests were performed on stand-alone Tiger and ChocoPy files of up to 31 766 characters (1 091 lines, 8 060 AST nodes), where 60% are smaller than a 10 000 characters. While performance seems linear in the size of the program, our algorithm might show different performance characteristics for much larger files or multiple dependent files. The performance of the algorithm is mostly governed by the performance of the Statix solver, which for larger files will have to track more variables in its unifier, and more nodes in the scope graph.

We were unable to test the algorithm on a language with a much larger static-semantic specification, but we expect the size of the program to have a bigger impact on performance. A larger specification mostly affects the number of rules that a predicate can be expanded into, whereas a larger program affects the size of the context that the Statix solver has to deal with.

The code completion algorithm is parameterized by the language's specification of static semantics in Statix. While van Antwerpen, Bach, et al. (2018b) and Rouvoet et al. (2020) show that many interesting name binding and type system concepts can be expressed in Statix, it is not clear what language concepts Statix cannot express and therefore to which languages this code completion approach does not apply.

3.7 Related Work

In this section, we compare our work to related work on code completion.

Code Completion in IDEs Editors for programming languages in IDEs such Eclipse[3] and IntelliJ IDEA[4] provide mature and sophisticated code completion services. However, implementing these editor services requires a considerable implementation effort in addition to the other aspects of the language implementation, an effort that has to be repeated for each IDE. Proposals such as the Language Server Protocol[5] (LSP) aim to reduce the extra effort required to support each editor, but do not offer more than a common interface for the code completion editor service, one that still requires a custom implementation for each language.

The value of our work is to provide a reliable code completion service without much additional work for the language engineer, making the effort of implementing the service reusable between languages. That is particularly interesting for language engineers working on new languages with a small team.

Textual Language Workbenches Language workbenches aim to support the implementation of rich editors for (domain-specific) programming languages. This usually includes support for implementing code completion, but typically requires language-specific implementation effort by the language engineer.

Xtext supports the generation of IDEs in Eclipse and other platforms (Xtext Team, 2021). Xtext's generates basic support for code completion, including reference proposals, from the grammar and scope providers. However, more sophisticated code completion needs to be implemented manually. Code completion does not take validation (type checking) rules into account. Such checks need to be built into code completion manually.

Rascal (Klint, van der Storm, and Vinju, 2009; Klint, van der Storm, and Vinju, 2010; Klint, van der Storm, and Vinju, 2019) provides TypePal, a framework for name analysis, type checking, and type inference (Rascal, 2021). TypePal provides useful utilities for writing type checkers and a name analysis representation. However, code completion cannot be derived automatically from such type checkers, but needs to be defined as a separate computation that takes the analysis result from the type checker as a parameter.

Spoofax (Kats and Visser, 2010b) supports sound and complete *syntactic* code completion by representing incompleteness through explicit placeholders and by synthesizing completion proposals from the grammar (de Souza Amorim, Erdweg, et al., 2016). They use placeholder inference to propose extensions of complete programs or to repair syntactic errors

[3] `https://www.eclipse.org/`
[4] `https://www.jetbrains.com/idea/`
[5] `https://microsoft.github.io/language-server-protocol/`

around the caret. We build on their work, restricting the syntactically correct proposals further by applying type checking and through name resolution in order to provide context-sensitive completion proposals. As future work, placeholder inference could be extended to propose (quick) fixes for static-semantic errors in programs.

Sasano and Choi (2021) use the *postfix sentential form* of a partial LR parse to determine candidates for code completion in a language-parametric way. However, they explicitly ignore any syntax following the caret that might restrict the possible candidates. And, as the title of the chapter suggests, they focus only on syntactic code completion; the types and scoping rules of the program are ignored, and identifiers are not proposed. Therefore, their approach might propose completions that introduce semantic errors into the program.

Structure Editors Structure editors or projectional editors operate on a structured representation via a projection of that representation. Representation of incomplete programs and completing such placeholders by means of code completion is an essential aspect of such editors.

The MPS language workbench is based on projectional editing (Jetbrains, 2021). A language is defined by means of a structure definition, characterizing the abstract structure of programs. An editor is created by defining projections and editing actions on structures. While MPS supports a rich set of projections (including graphical and diagram languages), it does not support automatic derivation of sound and complete synthesis of code completions from a language definition, including its static semantics. Edit actions in MPS are not guaranteed to be sound with respect to the type system rules of the language. That is, type constraints and other well-formedness constraints are not considered and need to be duplicated when synthesizing completion proposals.

Steimann, Frenkel, and Voelter (2017) address this issue in a prototype *robust projectional editor* extending MPS that supports *robust editing*, i.e., *editing that leaves a well-formed program well-formed.* This is realized by complementing the possibly unsound basic edits of MPS by proposing additional edits that repair the model to restore well-formedness. This can not only filter out invalid edits, but may also involve making non-local changes in the program. The RPE is realized by encoding well-formedness as a constraint satisfaction problem and computing edit operations using constraint-solving techniques. Because it would be too time-consuming to compute all possible edit scenarios, and too unwieldy for the user to use, the RPE has an interactive solution space explorer that starts with a few solutions and expands only upon specific user request. In this work, we focus on the more basic operation of code completions, but provide a predictable editor service. Furthermore, robust editing in the RPE can only be applied to entirely well-formed programs. In contrast, the Statix solver we use can deal with programs with errors; it solves as many constraints as

it can and report the failing constraints as errors. By extension, our code completion algorithm can synthesize sound completion proposals for the placeholder, by ignoring errors in the rest of the program.

Hazelnut (Omar, Voysey, Hilton, Aldrich, et al., 2017) is a structure editor based on a typed lambda calculus with holes, that is designed to study formal models and general principles of (structure) editing. This includes the construction of programs with locally type-incorrect terms through non-empty holes, which form a type barrier between an expected type and a provided type. The Hazelnut calculus formalizes edit actions to manipulate programs with (non-empty) holes, such that types are preserved. Subsequent work by Omar, Voysey, Chugh, et al. (2019) and Lubin et al. (2020) explore the execution and program synthesis of programs with holes. These are interesting avenues to explore. While we have not extended our language definitions with explicit non-empty holes, proposals such as `subterms.$LExp` in our running example are in fact similar. While `subterms` is not type-correct in the context of the placeholder, a possible 'continuation' could be. It would be useful to generalize such edit contexts.

Reusing Compiler Tools Our code completion algorithm is reusable across languages in the Spoofax language workbench. There are other approaches to reusable compiler tools or interfaces for them.

Luo, Dolby, and Bodden (2019) propose *MagpieBridge*, an approach to integrate static analysis in IDEs and editors similar to Language Server Protocol (LSP). They do this by integrating Soot, Doop, and WALA-based analysis through MagpieBridge with LSP, supporting any editor that supports LSP.

Söderberg and Hedin (2011) use *reference attribute grammars* (RAGs) in JastAdd to define semantic editor services, where large parts of existing RAG-based compilers can be reused. Their code completion service returns only 'accesses' (declarations), not syntax.

Pacak and Erdweg (2019) describe their vision for a semantic editor service by translating the typing rules to Datalog, and use existing Datalog solvers to not only verify the correctness of a program but also to answer 'code completion' queries: given a type, find expressions and variables of that type in scope. However, their translation into Datalog has to deal with infinite domains of types and typing contexts, and is not language-parametric.

Name Capture and Qualified Names Name capture concerns the introduction of a reference to a different declaration than intended. The *name-fix* algorithm of Erdweg, van der Storm, and Dai (2014) repairs name capture in code generators by computing a renaming of generated names. This kind of repair is not needed for our code completion, since our algorithm only computes names that are in scope, and thus are capture-free by construction. However, a limitation of our approach is that it will not directly propose names that

are shadowed, although the user can still reach them by inserting the qualifier through code completion, and then invoking code completion on the qualified placeholder.

In their work on identifier renaming for Java, Schäfer, Ekman, and de Moor (2008) introduce fixes for renamings that would lead to capture by qualifying names. For example, introducing the prefix `this.` can deconflict a reference to a class variable and a method parameter. Translating that approach would lead to proposing shadowed names if a deconflicting qualifier is available. It is an interesting future extension of our work to provide language-specific extensions to our language-parametric algorithm that provide such repairs.

Type Directed Programming In his work on Oleg, McBride (2000) developed a type theory with holes, placeholders for components of programs, to formalize programs under development. This work was later refined in the dependently typed language Epigram (McBride, 2005; McBride and McKinna, 2004). This has inspired the development of type-directed programming in subsequent dependently typed programming languages. Norell (2007) describes interaction points/holes in Agda, which piggyback on how Agda represents meta-variables internally, using a constraint-based approach based on McBride's treatment of meta-variables in Epigram. The language provides an Emacs integration with support for elaborating pattern match cases, summarizing and visualizing local bindings that could be used to fill the hole, and automatic code completion. The `auto` command of Agda performs a type-directed proof search trying to find a completion proposal considering constructors, local variables, and recursive calls, and picks the first candidate that fits (Lindblad and Benke, 2006). However, the `auto` command is neither sound nor complete. Similar support is provided by Idris (Brady, 2013) and Haskell.

It is an interesting challenge to apply our code completion algorithm to dependently typed languages. However, a complication of dependently typed programming languages is that the type checker requires normalization (computation) at compile-time. In principle, Statix can be used to specify a normalizer for type expressions. However, we have not defined a dependently typed language in Statix, let alone experimented with code completion for such a language.

Program Synthesis Semantic code completion is a very limited form of program synthesis, where the user intent is provided by the constraints imposed by the surrounding partial program, and the search is performed over possible solver states. The search strategy performs an enumerative search and checks whether each possible solution satisfies the program's semantic constraint (Gulwani, Polozov, and Singh, 2017).

While program synthesis systems such as Sketch (Solar-Lezama, 2009) can take the dynamic semantics of the code into account, our code completion algorithm only uses the

static semantics as specified by the language developer.

Program synthesis also requires the user to specify a specification, query, or constraints to find an appropriate solution, even when using top-down type-directed completion of partial expressions (Perelman et al., 2012). Static semantic code completion only needs the partial input program and the static-semantic specification of the language, without any additional user input.

Then again, contrary to most program synthesis, semantic code completion does not aim to find the best solution or even all possible solutions exhaustively. Instead, it performs a breadth-first search to find the immediately reachable (partial) solutions for the user to select from. This search is directed by the semantic rules of the language, and rule specializations in the language specification can be added to direct the search into certain directions for a more user-friendly experience without affecting the completeness or soundness of the proposals.

Machine Learning Asaduzzaman et al. (2014) and Raychev, Vechev, and Yahav (2014) use a large library of examples and histories to determine completion proposals and their ranking, using heuristics and statistical language models, respectively. While these approaches can be largely language independent, they are not sound, complete, or suitable for languages such as DSLs, that do not have a large code sample base to draw from. While we use a simple heuristic to rank proposals (most specific first), it would be useful to involve some sort of statistical model in this ranking, even in the face of small code bases or new languages.

Evaluation Hellendoorn et al. (2019) conclude that many synthetic benchmarks do not match real-world use of code completion. For example, real-world completion proposals were not clearly focused on any particular group of tokens. Also, they indicate that 'time saved', the time difference between the user finding the code completion proposal and typing the syntax or identifier manually, is often not a good measure, as the time between invoking code completion and accepting a proposal varies wildly. Their analysis is mostly geared toward code completion engines that use heuristics and statistics for their proposals. In this work, we have focused on a *correct-by-construction* derivation of code completion from language definitions. This is an important first step, but improving the presentation of code completion, in particular, the ranking as noted above, based on user evaluation, will be an important future work.

3.8 Future Work

In this section, we discuss some opportunities for improvements to the code completion algorithms usability or performance that do not impact its soundness or completeness.

Deterministic Expansion Without impacting the soundness or completeness, the code completion algorithm should perform *deterministic expansion* when there is only one possible expansion of a placeholder in a proposal. For example, for inserting a method call `foo(Int, Int)` that has two parameters, code completion currently suggests `foo($Exps)`, a method call with an argument list of indeterminate length. Deterministic expansion on `$Exps` would find that the only possible expansion is `$Exp, $Exp`, a list of two argument placeholders, and therefore code completion should immediately suggest `foo($Exp, $Exp)` instead.

We have an implementation of deterministic expansion, but for each placeholder in each proposal, it wasted a lot of time searching for a single expansion when there was none, and this had a significant negative impact on the overall performance. Therefore, we dropped it from this chapter, but one proposal is to amend the static-semantic specification with annotations that indicate when applying deterministic expansion makes sense.

Heuristics for Expansion We currently expand a predicate only once, unless it is a plain injection of the form `Stmt = Exp`. To manually force the algorithm to expand further, *rule specialization* (Section 3.4.5) can be used. However, we would like to identify a reliable heuristic that can automatically determine whether to continue expanding a proposal and when to stop. Similarly, when the proposal includes a placeholder that, when expanded several times more, could expand to a name, it would be better to present the user with this name than a placeholder.

Ranking Proposals Ranking proposals was out of the scope for this chapter, and therefore we ranked proposals by their specificity. However, we could employ statistical and/or example-based code completion techniques (Asaduzzaman et al., 2014; Asaduzzaman et al., 2016; Bruch, Monperrus, and Mezini, 2009; Raychev, Vechev, and Yahav, 2014), although their use might be hampered by the limited codebases available for certain DSLs.

Parallelization While our implementation of the code completion algorithm is single-threaded, one possible optimization is to run the search strategies in parallel. This would be made possible by the search states being immutable, and the search strategy not sharing mutable state. A related improvement could be to show the code completion proposals to the user as they are being discovered by the algorithm, while the algorithm continues exploring alternatives in the background.

Search Strategies The search strategies are not only useful for code completion, but potentially also in other language-parametric editor services that depend on the program's static analysis. In future work, we would like to investigate code generation, code navigation, structural search and replace, linting, and various refactorings. The search strategies could also be used to provide intelligent suggestions, such as relating parameter names in completion proposals to similarly named (and type compatible) variables in scope. Debuggers that display relevant information in the scope around a breakpoint could also benefit from search strategies to find, for example, the values of all variables in scope, or the members of a particular object.

3.9 Conclusion

We have presented a language-parametric code completion editor service that is sound and complete. It uses search strategies to find all type-safe proposals, including syntax, identifiers, and function calls. It does this language-parametrically for languages defined in the Spoofax language workbench, relying only on the Statix constraint solver and the existing syntax-directed static-semantic specification of the object language. Minimal effort is needed to produce a complete list of sound proposals, but small changes to the static-semantic specification can be used to direct code completion to produce more useful proposals. Through three steps of expanding predicates, injections, and queries, it attempts to find a set of useful proposals, one of which the user can select and insert at the caret location in the source code document. The algorithm rejects unsound proposals, and we demonstrated its completeness with respect to the static-semantic specification by having the algorithm reconstruct test files from scratch using repeated invocations of code completion. We have applied our algorithm to Tiger and ChocoPy for the tests in the evaluation, and Micro-C# for the examples in this chapter.

4

Language-Parametric Reference Synthesis

In the previous chapter, we showed how the static-semantic specification of a language can be used to implement code completion. Among the code completion proposals are names that are reachable from the caret position. In this chapter, we improve on this by using constraint solving to synthesize qualified names from the static-semantic specification. This can be used as a language-parametric building block for implementing refactorings. This chapter is a revised and extended version of the following paper, which was co-authored with Aron Zwaan and received a Distinguished Artifact Award:

> Daniel A. A. Pelsmaeker, Aron Zwaan, Casper Bach, and Arjan J. Mooij (2025a). "Language-Parametric Reference Synthesis." In: *Proceedings of the ACM on Programming Languages* 9.OOPSLA1, pp. 1213–1238. DOI: 10.1145/3720481

Abstract

Modern Integrated Development Environments (IDEs) offer automated refactorings to aid programmers in developing and maintaining software. However, implementing sound automated refactorings is challenging, as refactorings may inadvertently introduce name-binding errors or cause references to resolve to incorrect declarations. To address these issues, previous work by Schäfer et al. proposed replacing concrete references with locked references *to separate binding preservation from transformation. Locked references vacuously resolve to a specific declaration, and after transformation, must be replaced with concrete references that also resolve to that declaration. Synthesizing these references requires a faithful inverse of the name lookup functions of the underlying language.*

Manually implementing such inverse lookup functions is challenging due to the complex name-binding features in modern programming languages. Instead, we propose to automatically derive this function from type system specifications written in the Statix meta-DSL. To guide the synthesis of qualified references, we use scope graphs*, which represent the binding structure of a program, to infer their names and discover their syntactic structure.*

We evaluate our approach by synthesizing concrete references for locked references in 2528 Java, 196 ChocoPy, and 49 Featherweight Generic Java test programs. Our approach yields a principled language-parametric method for synthesizing references.

4.1 Introduction

> [P]reserving bindings is at the heart of any refactoring that moves, creates, or duplicates code. (Ekman, Schäfer, and Verbaere, 2008, §3)

As software projects evolve, their code is frequently refactored to improve their structure and maintainability. Refactoring often involves copying or moving code from one code unit (e.g., class, module, or trait) to another, in a way that preserves the program's behavior. A crucial aspect of behavior-preserving transformations is name-binding preservation, to ensure references in refactored code resolve to the same distinct declarations as before. While behavior preservation also needs control- and data-flow analysis, name-binding preservation can be achieved using only the static-semantic analysis of the program. However, due to the sophisticated name-binding features found in many modern programming languages, preserving the name-resolution semantics of code across transformations is generally challenging.

To illustrate the complexity of reasoning about advanced name-binding features, consider the Java program shown in Fig. 4.1a. If we rename the field `x` (line 2) to `y`, Java's static semantics would cause the reference to `y` on line 7 (in method `foo`) to resolve to the newly

renamed field `y` on line 2, rather than the intended declaration of `y` on line 5. This undesired change would alter the name-binding structure of the program. To prevent this, the reference to `y` on line 7 should be *qualified* as `Outer.this.y`, as shown in the refactored example in Fig. 4.1b.

Transformations that require name-binding preservation are common across many refactorings, such as those from Fowler's catalog (Fowler, 1999). Yet, manually refactoring code is time-consuming and error-prone. Consequently, many modern Integrated Development Environments (IDEs) provide automated refactorings such as Rename, Inline/Extract Method, and Pull Up/Push Down (Fowler, 1999), which attempt to automatically *requalify* references to maintain the program's binding structure.

However, even popular IDEs for mainstream languages struggle to implement sound refactorings. For example, Ekman, Schäfer, and Verbaere (2008) identify several bugs in Eclipse 3.4, where automated refactorings inadvertently altered the program's binding structure. These errors arise from the difficulty of accurately determining which references need to be fixed and computing the correct requalifications. Not only references in the modified code, but references *throughout the entire codebase* may require requalification. Ensuring both *soundness* (preserving name bindings) and *completeness* (finding all possible requalifications) is particularly difficult.

Ekman, Schäfer, and Verbaere conclude that these challenges are "not related to the core ingredients of the implemented refactoring, [but] inherent to the complexity of name-binding rules in mainstream languages." As a result, existing research on the sound requalification of references is often language-specific, focusing on mainstream languages like Java (Schafer et al., 2012). Implementing sound automated refactorings for other languages, like domain-specific programming languages (DSLs) with small language developer teams, can require a prohibitively high effort. As such, a more principled and language-parametric approach to guarantee name-binding preservation is needed.

4.1.1 Locked References

Ekman, Schäfer, and Verbaere (2008) observe that many bugs in automated refactorings could "be avoided if a set of carefully crafted building blocks were available to refactoring developers." One such building block is *locked references*[1], proposed by Schäfer et al. in previous work (Schäfer, Ekman, and de Moor, 2008; Schaefer and de Moor, 2010). A locked reference is an abstract reference that continues to refer to the same unique declaration even

[1]Terminology introduced by Schafer et al. (2012). Also referred to as "bound names" (Schäfer, Verbaere, et al., 2009), "locked names" (Schaefer and de Moor, 2010), and "locked bindings" (Schafer et al., 2012). We use "locked references" throughout this chapter.

```
1 class Base {
2   int x = 1;
3 }
4 class Outer {
5   int y = 2;
6   class Inner extends Base {
7     int foo() { return y; }
8   }
9 }
```

(a) Before renaming.

```
1 class Base {
2   int y = 1; // x renamed to y
3 }
4 class Outer {
5   int y = 2;
6   class Inner extends Base {
7     int foo() { return Outer.this.y; }
8   }
9 }
```

(b) After renaming.

Figure 4.1: RENAME refactoring of a small Java program, where renaming the field `x` to `y` on line 2 requires the reference `x` on line 7 to be appropriately qualified.


```
1 class Base {
2   int x1 = 1;
3 }
4 class Outer {
5   int y2 = 2;
6   class Inner extends Base {
7     int foo() { return (|→y2|); }
8   }
9 }
```

(a) After locking references, before renaming.

```
1 class Base {
2   int y1 = 1; // x renamed to y
3 }
4 class Outer {
5   int y2 = 2;
6   class Inner extends Base {
7     int foo() { return (|→y2|); }
8   }
9 }
```

(b) After renaming, before unlocking references.

Figure 4.2: Intermediate steps for performing the RENAME refactoring from Fig. 4.1 using locked references. After locking the relevant reference `y` to declaration y_2 (a) and performing the transformation (b), our approach would synthesize a solution for the locked reference and obtain Fig. 4.1b.

if code is moved or the declaration is renamed. This ensures that transformations cannot cause such a reference to accidentally capture a different declaration.

The following diagram summarizes program transformation with locked references:

$$\mathscr{P} \xrightarrow{\text{lock}} \mathscr{P}_{\text{locked}} \xrightarrow{\text{transform}} \mathscr{P}'_{\text{locked}} \xrightarrow{\text{unlock}} \mathscr{P}'$$

Before refactoring, we first 'lock' each relevant concrete reference by replacing it with a locked reference pointing to the original declaration. In Fig. 4.2a, we replace the concrete reference `y` (line 7) with a locked reference $(|{\to}y_2|)$ to the declaration y_2 on line 5.[2] (We use subscript indices to distinguish different occurrences of the same name, but the indices are not part of the syntax.)

Next, we 'transform' the program as required for the refactoring, renaming declarations and moving code. In Fig. 4.2b, the declaration on line 2 is renamed to `y`. Finally, we 'unlock'

[2] Our syntax for locked references $(|{\to}d|)$ is inspired by the syntax Omar, Voysey, Hilton, Aldrich, et al. (2017) use for holes.

each locked reference in the program by replacing it with a *synthesized* concrete reference that unambiguously resolves to the intended declaration. In this case, unlocking replaces the locked reference with `Outer.this.y`, maintaining the name-binding semantics of the program (see Fig. 4.1b).

Every step in this pipeline gives rise to challenges, but in this chapter, we focus on the key challenge of synthesizing concrete references when unlocking locked references. The program should remain well-typed, and synthesized references should resolve to their intended declarations. Separating name-binding preservation from the transformation guarantees that refactorings preserve name bindings, and also simplifies refactorings.

There are numerous potential applications of reference synthesis. In the line of work by Schäfer, Ekman, and de Moor (2008) and Schafer et al. (2012), it can be applied to implement sound (editor) refactorings. Furthermore, it provides a powerful transformation tool for implementing sound transformations of DSL programs or performing large-scale codebase transformations aiming to improve the overall code quality. However, one can also envision *user-extensible refactoring tools* (such as those presented by Li and Thompson (2012)) or *transformation languages* (such as IntelliJ IDEA's structural search and replace) that need to preserve name bindings. Finally, it could be used to build an editor service that suggests fixes for type errors (e.g., QUICK FIX in Eclipse).

4.1.2 Language-Parametric Reference Synthesis

Following Schäfer, Ekman, and de Moor (2008) and Schafer et al. (2012), a reference synthesis function can be thought of as the right inverse of a reference resolution function. That is, if *QRef* is the set of qualified references, *Decl* is the set of uniquely identified declarations, and the function $\mathsf{resolve}_p : \mathit{QRef} \rightharpoonup \mathit{Decl}$ resolves a reference at some location p in the program, then locked reference synthesis should be a function $\mathsf{synthesize}_p : \mathit{Decl} \rightharpoonup \mathit{QRef}$ such that $\mathsf{resolve}_p(\mathsf{synthesize}_p(d)) = d$ for any p.[3] When replacing a concrete reference with a locked reference, the lock function uses resolve to obtain the target declaration, and conversely, unlock uses synthesize to replace the locked reference with a concrete reference.

The reference synthesis pipeline shown above is conceptually *language-parametric*. However, as discussed before, implementing correct reference synthesizers manually is error-prone and time-consuming. In this chapter, we present a *language-parametric* approach to derive the synthesize function automatically from declarative type system specifications, letting language designers generically synthesize valid concrete references for programs with multiple locked references. The goal of reference synthesis is to find for each locked

[3] Schäfer, Ekman, and de Moor use "lookup" instead of "resolve" and "access" instead of "synthesize," but the idea is the same.

reference in a program a valid concrete reference that resolves to the intended declaration.

We derive the synthesize function from *only* a declarative specification of the language's static semantics, specified using the *Statix* specification language (van Antwerpen, Bach, et al., 2018b; Rouvoet et al., 2020). Statix allows syntax-directed typing rules to be specified, uses *scope graphs* as a declarative model of static name binding and name resolution (Neron et al., 2015), and generates executable type checkers by interpreting specifications as constraint programs. In our implementation of synthesize, we reinterpret the *name-resolution queries* from the specification to infer syntax for concrete references that resolve to a particular target declaration, guaranteeing name-binding preservation. We reuse the Statix solver (see Section 4.2) to validate that the syntax we infer is *sound* with respect to the typing rules and represents a reference that resolves to the intended declaration.

This chapter makes the following technical contributions:

- We present a language-parametric implementation of the synthesize function (see Section 4.4 and Section 4.5). This function automatically synthesizes concrete references that resolve to the specified declaration, and that are sound with respect to a type system specification written in Statix.

- We evaluate our implementation on 2773 test programs of Java, ChocoPy, and Featherweight Generic Java (Section 4.6). Our results demonstrate that our approach applies to mainstream languages with complex name-binding semantics without modifying their typing rules.

We first (Section 4.2) introduce scope graphs and Statix. Next, in Section 4.3 we illustrate our reference synthesis algorithm. Then, in Section 4.4 we give an operational semantics of our synthesize function's implementation, and the heuristics we apply in Section 4.5. We evaluate our implementation in Section 4.6, and discuss related work in Section 4.7. We conclude in Section 4.8 and discuss future work in Section 4.9.

4.2 Scope Graphs and Statix

This chapter presents a *language-parametric* approach to synthesizing concrete references. To this end, we build on existing work: (1) *scope graphs* as a language-parametric model of name binding, and (2) *Statix* as a uniform representation of typing rules.

In this section, we first describe what scope graphs are (Section 4.2.1) and how they let us resolve references via graph search (Section 4.2.2). Then we provide a high-level introduction to the Statix language (Section 4.2.3) and its constraint solver (Section 4.2.4).

4.2.1 Scope Graphs

The example program in Fig. 4.3a contains two declarations named x, namely x_1 on line 1 and x_2 on line 3, and a named reference x on line 7.[4] The question is: does x refer to declaration x_1 or x_2? Either can be true, depending on the semantics of the programming language.

Scope graphs (Neron et al., 2015; Rouvoet et al., 2020; van Antwerpen, Bach, et al., 2018b; van Antwerpen, Néron, et al., 2016; Zwaan and van Antwerpen, 2023) offer a uniform model for name resolution that supports sophisticated name-binding patterns in programming languages. As their name suggests, scope graphs model the scoping structure of programs as graphs. Such graphs let us model both nested and recursive scoping structures, and name-resolution policies as graph search queries.

To illustrate this, consider the program and its scope graph in Fig. 4.3. The nodes in the graph represent scopes: s_0 represents the *global scope*, while s_A and s_B represent the scopes of modules A and B, respectively. Scopes s_{x1}, s_{x2}, and s_y represent named declarations. A scope s may have data d associated with it, written as $s \mapsto d$, such as the name of the module that they correspond to or the name and type of their declaration. As we shall see later, associating scopes with names lets us define *name-resolution queries* that resolve module names to their corresponding scopes.

Edges between scopes represent *reachability* relations. Queries can follow these edges to reach other scopes and declarations. The two module scopes are reachable from the global scope via MOD edges, representing the fact that A and B are modules declared in the global scope s_0. The module scopes s_A and s_B are also lexical children of the global scope, so each is connected via a LEX edge to s_0. VAR edges connect scopes to declared names. Due to the wildcard `import A::*` on line 6 (which imports all members of the module A) module scope s_B is connected to module scope s_A via an IMP edge, making all declarations in module A reachable from module B.

4.2.2 Scope Graph Queries

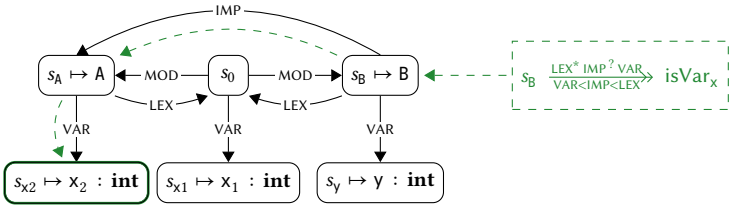
We define name resolution as *queries* in scope graphs. Resolving a query entails finding all paths from this source scope to matching declarations. To explain the syntax of a name-resolution query, we take the query shown in the dashed box on the right of the graph in Fig. 4.3b:

$$s_B \xrightarrow[\text{VAR}<\text{IMP}<\text{LEX}]{\text{LEX}^* \text{ IMP}^? \text{ VAR}} \twoheadrightarrow \text{isVar}_x$$

[4] We use *subscript indices* to distinguish different *occurrences* of a particular name. For example, x_2 uniquely identifies one of the declarations named x.

```
var x1 = 42
mod A {
  var x2 = 0
}
mod B {
  import A::*
  var y = x
}
```

(a) Program.



(b) Scope graph for the program in Fig. 4.3a.

Figure 4.3: An example LM (Neron et al., 2015) program and its scope graph, where boxes and arrows represent scopes and reachability relations between scopes, respectively. The dashed box represents a query and the dashed arrows its resolution path to x_2, the second occurrence of a declaration named x.

Here, s_B is the initial scope of the graph search, and isVar_x is a filter that ensures only declarations with name x are selected. The regular expression $\text{LEX}^* \text{ IMP}^? \text{ VAR}$ is a *reachability policy* declaring which declarations are reachable; i.e., those declarations we can reach by following a sequence of labeled edges that match the regular expression. The path ordering $\text{VAR} < \text{IMP} < \text{LEX}$ is a *visibility policy* used to disambiguate which reachable names are visible, i.e., to model shadowing. For example, both s_{x1} and s_{x2} are reachable in Fig. 4.3b. However, the order prefers IMP edges over LEX edges, so the only valid path through the graph is the path to s_{x2}.

4.2.3 Statix Rules and Constraints

In classical typing rules, terms are typed relative to one or more *typing contexts* (Pierce, 2002), or typed via *symbol tables* (Aho and Ullman, 1972) or *class tables* (Igarashi, Pierce, and Wadler, 2001). Following existing work (van Antwerpen, Bach, et al., 2018b; Bach et al., 2018; Rouvoet et al., 2020), we can define typing rules in a similar style, but with terms typed relative to one or more scopes in a scope graph instead. The constraint language Statix (van Antwerpen, Bach, et al., 2018b; Rouvoet et al., 2020; van Antwerpen and Visser, 2021b) lets us declare such inference rules using a syntax inspired by logic programming. Type system specifications written in Statix have a declarative interpretation, specifying a class of well-typed programs. Alternatively, specifications can be used operationally to type-check programs by constructing a scope graph and resolving references by traversing it. This subsection highlights the main features of Statix rules and constraints. For a more detailed breakdown of the syntax, we refer to the discussion in Section 4.4.1 and the work of Rouvoet et al. (2020).

The rules in Fig. 4.4 show a representative subset of the Statix rules we derived for LM, a toy language from Neron et al. (2015) used throughout this chapter. The figure declares

$$\mathsf{typeOfExpr}(s, n, \mathbf{int}) \leftarrow \mathsf{emp} \quad \text{(T-Num)}$$

$$\mathsf{typeOfExpr}(s, e_1 + e_2, \mathbf{int}) \leftarrow \mathsf{typeOfExpr}(s, e_1, \mathbf{int}) * \mathsf{typeOfExpr}(s, e_2, \mathbf{int}) \quad \text{(T-Add)}$$

$$\mathsf{typeOfExpr}(s, x, T) \leftarrow s \xrightarrow[\text{VAR}<\text{IMP}<\text{LEX}]{\text{LEX}^*\,\text{IMP}^?\,\text{VAR}} \mathsf{isVar}_x \mapsto \{(_, x : T)\} \quad \text{(T-Var)}$$

$$\begin{aligned}\mathsf{typeOfExpr}(s, a.x, T) \leftarrow\ & \exists s_m.\ \mathsf{scopeOfMod}(s, a, s_m) * \\ & s_m \xrightarrow{\text{VAR}} \mathsf{isVar}_x \mapsto \{(_, x : T)\}\end{aligned} \quad \text{(T-QRef)}$$

$$\begin{aligned}\mathsf{memberOk}(s, \texttt{var}\ x = e) \leftarrow\ & \exists T\, s_x.\ \mathsf{typeOfExpr}(s, e, T) * \\ & \nabla s_x \mapsto x : T * s \xrightarrow{\text{VAR}} s_x\end{aligned} \quad \text{(M-Var)}$$

$$\begin{aligned}\mathsf{modOk}(s, \texttt{mod}\ a\ \{\ \overline{imp}\ \overline{mem}\ \}) \leftarrow\ & \exists s_m.\ \nabla s_m \mapsto a * s_m \xrightarrow{\text{LEX}} s * s \xrightarrow{\text{MOD}} s_m * \\ & \mathsf{importsOk}(s_m, \overline{imp}) * \mathsf{membersOk}(s_m, \overline{mem})\end{aligned} \quad \text{(M-Mod)}$$

$$\mathsf{importOk}(s, \texttt{import}\ a::*) \leftarrow \exists s_m.\ \mathsf{scopeOfMod}(s, a, s_m) * s \xrightarrow{\text{IMP}} s_m \quad \text{(D-ImportOk)}$$

$$\mathsf{scopeOfMod}(s, x, s_m) \leftarrow \exists p.\ s \xrightarrow[\text{MOD}<\text{LEX}]{\text{LEX}^*\,\text{MOD}} \mathsf{isMod}_x \mapsto \{(p, x)\} * s_m \stackrel{?}{=} \mathsf{tgt}(p) \quad \text{(S-Mod)}$$

$$\begin{aligned}\mathsf{scopeOfMod}(s, a.x, s_m) \leftarrow\ & \exists p\, s'_m.\ \mathsf{scopeOfMod}(s, a, s'_m) * \\ & s'_m \xrightarrow{\text{MOD}} \mathsf{isMod}_x \mapsto \{(p, x)\} * s_m \stackrel{?}{=} \mathsf{tgt}(p)\end{aligned} \quad \text{(S-QMod)}$$

$$\mathsf{isVar}_x \triangleq \lambda x'.\ \exists T.\ (x : T) \stackrel{?}{=} x'$$

$$\mathsf{isMod}_x \triangleq \lambda x'.\ x \stackrel{?}{=} x'$$

Figure 4.4: A subset of the typing rules of LM, a toy language from (Neron et al., 2015) used for the examples in this chapter.

rules for five different typing relations: typeOfExpr, memberOk, modOk, importOk, and scopeOfMod. Each rule has a conclusion on the left of an arrow (←), and a premise given by one or more constraints on the right. For example, the rule T-Add states that the expression $e_1 + e_2$ has type **int** in scope s, if both e_1 and e_2 have type **int** under the same scope s. Rule T-QRef is a more complex example, where a qualified module access expression $a.x$ has type T when the a resolves to a module (asserted by the predicate constraint $\mathsf{scopeOfMod}(s, a, s_m)$), and x resolves from that module to a declaration of type T (asserted by the query constraint $s_m \xrightarrow{\text{VAR}} \mathsf{isVar}_x$).

The rules in Fig. 4.4 do not mention the underlying scope graph explicitly. Instead, premises of rules assert requirements on the scope graph structure, such as the existence of scopes with associated data ($\nabla s_x \mapsto x : T$) and edges ($s \xrightarrow{\text{VAR}} s_x$) and the ability to resolve query constraints. A program is well-typed when a *minimal* scope graph exists that satisfies each such assertion and query. Minimality implies that the scope graph only has the scopes and edges asserted by the rules of a program: no extraneous edges or scopes. There exists a solver for Statix constraints that computes this minimal scope graph, which we discuss in the next section.

4.2.4 Statix Constraint Solver

Following Rouvoet et al. (2020), the operational semantics of Statix is given by a constraint solver that soundly constructs and queries scope graphs, and uses unification to solve equality constraints. The reference synthesis approach we illustrate in Section 4.3 is sound by construction because it builds on this operational semantics. We defer a deeper discussion of the operational semantics to Section 4.4.

The Statix solver will solve as many constraints as possible, yielding either a state with no unsolved constraints (i.e., the program type-checks), a state that derives false (i.e., the program does not type-check), or a *stuck* state, where the solver does not have enough information to solve the remaining constraints. There are two reasons why a constraint can get stuck: either (1) it is not sufficiently instantiated, or (2) it is a query constraint that is not yet guaranteed to yield a *stable answer*. For (1), the solver will only expand a predicate such as $\mathsf{typeOfExpr}(x, y, z)$ once x, y, and z are sufficiently instantiated such that only a single rule matches. Similarly, it will only run and solve a query constraint once its *source scope* (e.g., s in $s \xrightarrow[\mathsf{VAR}<\mathsf{IMP}<\mathsf{LEX}]{\mathsf{LEX}^*\,\mathsf{IMP}^?\,\mathsf{VAR}} \mathsf{isVar}_x$) and *data well-formedness predicate* (e.g., isVar_x) are ground. In case (2), a query gets stuck when it needs to run, but another unsolved constraint might add a scope graph edge that could invalidate the query. The Statix solver implements guards that detect these cases (Rouvoet et al., 2020).

Our synthesize function runs the Statix solver on a program with holes, where each hole is represented by a free unification variable that maps to a target scope representing the hole's intended target declaration. The unification variables cause the Statix solver to get stuck on predicate and query constraints directly related to the holes. Once a stuck state is reached, our reference synthesis approach extends the usual operational semantics of Statix with the ability to use the typing rules of a language to refine the holes of the program, and use the Statix solver to verify the solution. Once the term of a hole becomes ground and all constraints in the state have been solved, we have successfully synthesized a concrete reference.

We will illustrate how the Statix solver, scope graph, and typing rules are used in our reference synthesis algorithm in the next section, and discuss the operational semantics of Statix and our extension in more detail in Section 4.4.

4.3 Reference Synthesis by Example

As the name suggests, *reference synthesis* is used to synthesize a concrete (qualified) *reference* to a given declaration. References in many languages take the shape $x_1.x_2.\cdots.x_n$, modulo syntax. Here, the first name x_1 is resolved from the place in the program where the reference occurs, and subsequent names x_i are resolved relative to wherever the previous qualifiers $x_1.\cdots.x_{i-1}$ led. The final name x_n leads to the target declaration.

This informal definition of a reference encompasses many syntactic constructs that we intuitively recognize as (qualified) references across languages, for example, `ID IN CUSTOMER IN LAST-TRANSACTION` in COBOL, `Person.this.name` in Java, and `std::option::Option` in Rust. On the other hand, according to our definition, syntax like `List<String>` in Java does not constitute a reference: it is a parameterized type, akin to how a method call `foo(x, y)` would also not be considered a reference. We give a more precise definition of a reference in Section 4.4.

$$\mathscr{P} = X[\overline{r}] \xrightarrow{\text{lock}} X[\,\overline{(\!|\to d|\!)}\,] \xrightarrow{\text{transform}} X'[\,\overline{(\!|\to d|\!)}\,] \xrightarrow{\text{unlock}} X'[\overline{r'}] = \mathscr{P}'$$

The above diagram reiterates program transformation with locked references. Initially, we have a program $\mathscr{P}$, which can be represented as a context[5] X with references $\overline{r}$ occurring in it. First, we lock relevant references. That is, we replace concrete references with locked references that durably remember which declaration they point to, regardless of where the locked reference occurs in the program ($X[\,\overline{(\!|\to d|\!)}\,]$). Then we transform the program, possibly moving code around ($X'[\,\overline{(\!|\to d|\!)}\,]$). Finally, we unlock the locked references: synthesizing their concrete references ($\overline{r'}$) and plugging them into the program to yield the concrete transformed program ($\mathscr{P}' = X'[\overline{r'}]$).

In this section, we use a simple program with a locked reference, shown in Fig. 4.5, to illustrate the semantics of our synthesize function. The idea is to model locked references as holes given by unification variables, and strategically apply typing rules to infer a substitution for each hole. The strategy used to apply typing rules must guarantee that inferred substitutions correspond to references that resolve as intended.

4.3.1 Initial Constraint Solving

Consider the example in Fig. 4.5a, where for the locked reference $(\!|\to \mathtt{y}|\!)$ we want to synthesize a concrete reference that must resolve to variable `y`'s declaration scope in the underlying scope graph. Valid concrete references that our approach could yield include `y` and `A.y`.

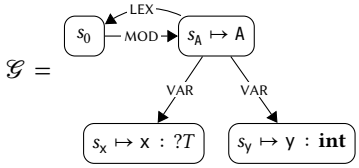
[5] We can regard a 'context' as a zipper-like structure over an AST.

```
mod A {
  var x = ⟪→y⟫
  var y = 1
}
```

(a) Program with locked reference.

```
mod A {
  var x = ?E
  var y = 1
}
```

(b) Program with hole.



(c) Scope graph.

Figure 4.5: Small example program and its scope graph.

The first step of our approach is to replace each locked reference in the program by a hole, represented by a fresh unification variable, shown in Fig. 4.5b. Then we use the original language's static-semantic rules and run the Statix solver on the input program. The presence of holes causes the solver to yield a stuck state, where the solver neither has enough information to solve all constraints, nor can derive false, due to the free unification variables.

In our example, the Statix solver will recursively expand predicates and solve scope graph constraints to yield a solver state, shown below, with the inferred scope graph $\mathscr{G}$ (see Fig. 4.5c) and a single constraint that is stuck because Statix cannot infer which rule to apply to expand the predicate. Additionally, in the state we record that the unification variable $?E$ is associated with hole h, and that hole h should become a concrete reference that resolves to the scope s_y, which is the scope associated with the locked reference's target declaration y in $\mathscr{G}$.

$$\left\langle \mathscr{G} \;\middle|\; \text{typeOfExpr}(s_A, ?E, ?T) \;\middle|\; ?E \mapsto h \;\middle|\; h \mapsto (s_y, ?E) \right\rangle$$

The state has the form $\langle \mathscr{G} \,|\, \overline{C} \,|\, U \,|\, H \rangle$. Following Rouvoet et al. (2020), the solver state is given by the (partially constructed) scope graph $\mathscr{G}$ and the set of yet unsolved constraints $\overline{C}$. To support reference synthesis, we have augmented the solver state with U and H. U is a partial function that maps free unification variables to hole identifiers where applicable. H maps each hole identifier in a program to a pair (s_t, t) of the current target scope s_t and the term t synthesized for the hole so far. Note that s_t changes as we synthesize qualifiers for the concrete reference.

4.3.2 Forking States

By default, the Statix solver only expands a constraint when there is exactly one possible expansion, and otherwise the constraint gets stuck. To support reference synthesis, we augment the solver to allow solver states to be *forked*. This way, we can speculatively apply

each possible expansion of a constraint, obtaining a solver state. Forked solver states that fail are discarded, but those that can be successfully solved represent programs for which we have synthesized a valid reference.

For the stuck state discussed above, we can speculatively expand the stuck typeOfExpr predicate constraint and fork the state for each of the typeOfExpr rules shown in Fig. 4.4, yielding different states. However, some of those rules will not lead to well-formed references, and therefore, we only expand to rules that could yield a reference. For the simple LM language shown in Fig. 4.4, the relevant rules are T-Var and T-QRef. We fork the solver state, and discuss how each of these two rules leads to a synthesized reference.

4.3.3 Expanding Query Constraints

Applying the rule T-Var and solving the constraints as far as possible yields the first forked solver state with one stuck query constraint:

$$\left\langle \mathcal{G} \;\middle|\; s_{\mathsf{A}} \xrightarrow[\mathsf{VAR}<\mathsf{IMP}<\mathsf{LEX}]{\mathsf{LEX}^*\,\mathsf{IMP}^?\,\mathsf{VAR}} \!\!\gg \mathsf{isVar}_{?x} \mapsto \{(_, ?x : ?T)\} \;\middle|\; ?x \mapsto h \;\middle|\; h \mapsto (s_{\mathsf{y}},\ ?x\ ) \right\rangle$$

As the free variable $?x$ in the stuck constraint is related to hole h, we can infer that the query is also related to hole h: attempting to solve the constraint could be fruitful for synthesizing a reference to the target scope s_{y}. Therefore, reference synthesis searches for valid scope graph paths from s_{A} to the intended target scope s_{y}, while respecting the reachability regex and visibility ordering of the query. There is exactly one such path, namely the one-step path traversing the VAR edge from s_{A} to s_{y}. This implies that $?x = \mathsf{y}$ and $?T = \mathbf{int}$. This solves all constraints and results in the mapping $h \mapsto (s_{\mathsf{y}}, \mathsf{y})$. As all constraints have been solved and the term for the hole is ground, the unqualified reference y is returned as a solution.

4.3.4 Qualified Reference

In the second forked solver state, applying the rule T-QRef and solving constraints yields the following solver state with a stuck predicate constraint and a stuck query constraint, where the term for the hole represents a possible *qualified* reference $?a.?x$:

$$\left\langle \mathcal{G} \;\middle|\; \begin{array}{c} \mathsf{scopeOfMod}(s_{\mathsf{A}}, ?a, ?s_m) \\ ?s_m \xrightarrow{\mathsf{VAR}} \!\!\gg \mathsf{isVar}_{?x} \mapsto \{(_, ?x : ?T)\} \end{array} \;\middle|\; \begin{array}{l} ?a \mapsto h \\ ?x \mapsto h \end{array} \;\middle|\; h \mapsto (s_{\mathsf{y}},\ ?a.?x\ ) \right\rangle$$

Next, we expand the scopeOfMod predicate. One of the possible expansions (using rule S-Mod) yields the following state:

$$\left\langle \mathcal{G} \;\middle|\; \begin{array}{ll} s_{\mathsf{A}} \xrightarrow{\text{LEX}^* \text{ MOD}}\!\!\!\!\rightarrow \text{isMod}_{?a} \mapsto \{(?p, ?a)\} & (1) \\ ?s_m \stackrel{?}{=} \text{tgt}(?p) & (2) \\ ?s_m \xrightarrow{\text{VAR}}\!\!\!\!\rightarrow \text{isVar}_{?x} \mapsto \{(_, ?x : ?T)\} & (3) \end{array} \;\middle|\; \begin{array}{l} ?a \mapsto h \\ ?x \mapsto h \end{array} \;\middle|\; h \mapsto (s_{\mathsf{y}}, ?a.?x) \right\rangle$$

The solver state has two stuck query constraints: the module query (1) and the variable query (3). Both query constraints have unification variables that relate to the hole h, so we cannot know which of these queries should resolve to the target scope s_{y}. Therefore, we fork the solver state again: one branch where we attempt to expand query (1) and one where we attempt to expand query (3). Only the fork that attempts to expand the variable query (3) to the target s_{y} will succeed, so in this example we continue with that branch.

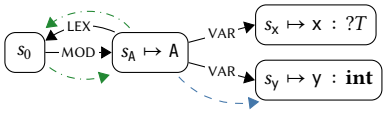
Because of the mapping $?x \mapsto h$, query (3) may be relevant for resolving to the target scope s_{y}. Thus, we inspect the scope graph and search for well-formed paths to s_{y}. Since the query in question has the unification variable $?s_m$ as its source, we need to look for paths with *any* possible source. For the graph Fig. 4.5c in the solver state, only the one-step path from s_{A} to s_{y} matches the regular expression of the query. Hence, the query is resolved by substituting s_{A} for the scope variable $?s_m$. Next, we make s_{A} the new target for the hole h, since we assume that the source scope was not ground because the query forms part of a qualified reference; i.e., a sequence of paths. Eventually, we solve the constraint by substituting $?s_m \mapsto s_{\mathsf{A}}$ and $?x \mapsto \mathsf{y}$.

$$\left\langle \mathcal{G} \;\middle|\; \begin{array}{l} s_{\mathsf{A}} \xrightarrow{\text{LEX}^* \text{ MOD}}\!\!\!\!\rightarrow \text{isMod}_{?a} \mapsto \{(?p, ?a)\} \\ s_{\mathsf{A}} \stackrel{?}{=} \text{tgt}(?p) \end{array} \;\middle|\; ?a \mapsto h \;\middle|\; h \mapsto (s_{\mathsf{A}} \cdot s_{\mathsf{y}}, ?a.\mathsf{y}) \right\rangle$$

In addition to refining the hole term to $?a.\mathsf{y}$, the target scope was also refined to s_{A}. The new problem to be solved is to find the qualifier that resolves to s_{A}. Using the same principles as illustrated above, the remaining stuck query can be expanded. This will solve the remaining constraints and make the hole term ground, yielding `A.y` as the solution.

$$\left\langle \mathcal{G} \;\middle|\; \varnothing \;\middle|\; \varnothing \;\middle|\; h \mapsto (s_{\mathsf{A}} \cdot s_{\mathsf{A}} \cdot s_{\mathsf{y}}, \mathsf{A.y}) \right\rangle$$

Following our definition of a reference, the solution `A.y` is a composite path in the scope graph from the source scope s_A, $s_A \dashrightarrow s_0 \dashrightarrow s_A$ to the scope of qualifier `A`, followed by $s_A \dashrightarrow s_y$ to the scope of the intended target declaration `y`, as shown here:



The next section formally defines the approach illustrated above. In Section 4.5 we describe the heuristics we apply to make the approach usable in practice. We evaluate our synthesize function on test programs with locked references in Section 4.6.

4

4.4 Operational Semantics

The previous section illustrated our approach to synthesize concrete references using an extension of the Statix solver. This section presents an operational semantics that defines that extension.

4.4.1 Syntax of Statix

The syntax of Statix terms and constraints, defined in Fig. 4.6, follows Rouvoet et al. (2020) and has a separation-logic-inspired flavor, as the declarative semantics of Statix constraints is defined using separation logic. We refer to Rouvoet et al. for the details of this declarative semantics, and focus on the operational semantics instead in Section 4.4.2 and Section 4.4.3.

The syntax uses these distinct enumerable sets of symbols: *TermConstructor* for term constructor symbols, *Var* for term variables, *SetVar* for set variables, *PredSymbol* for names of predicates (such as typeOfExpr in Fig. 4.4), *Scope* for scope graph node identifiers, *Label* for scope graph edge labels, *RegEx* is the set of regular expressions over words comprised of label symbols, and *PartialOrd* is the set of partial orders on label symbols.

In the *Constraint* syntax, emp is the trivially satisfiable constraint, akin to a true constraint in a traditional logic. Conversely, false is never satisfiable. $C_1 * C_2$ is a *separating conjunction*, where the declarative and operational semantics of Statix guarantee C_1 and C_2 construct separate scope graph fragments. However, the reader can approximately think of $C_1 * C_2$ as traditional logic conjunction. The constraint $\exists x.\, C$ asserts the existence of some term named x, which may be referenced and constrained by C. $\mathsf{single}(t, \underline{t})$ asserts that set term $\underline{t}$ is a singleton set whose element is equal to t, while $\forall x \text{ in } \underline{t}.\, C$ asserts that C holds for all its elements x. $\nabla t_1 \mapsto t_2$ asserts that the scope graph contains a scope identified by t_1 and with associated data t_2, whereas $t_1 \xrightarrow{l} t_2$ asserts that the scope identified by t_1 is connected via an l-labeled edge to the scope identified by t_2.

$$
\begin{array}{rcl}
f \in \mathit{TermConstructor} & \quad & s \in \mathit{Scope} \\
x \in \mathit{Var} & & l \in \mathit{Label} \\
z \in \mathit{SetVar} & & r \in \mathit{RegEx} \\
P \in \mathit{PredSymbol} & & o \in \mathit{PartialOrd}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{Term} \ni t & ::= & x \mid f(t^*) \mid l \mid s \\
\mathit{SetTerm} \ni \underline{t} & ::= & z \mid \zeta \\
\mathit{SetLit} \ni \zeta & ::= & \varnothing \mid \{t\} \mid \zeta \sqcup \zeta \\
\mathit{Constraint} \ni C & ::= & \mathsf{emp} \mid \mathsf{false} \mid C * C \mid \exists x.\, C \mid \mathsf{single}(t, \underline{t}) \mid \forall x \text{ in } \underline{t}.\, C \\
& \mid & \nabla t \mapsto t \mid t \xrightarrow{l} t \mid t \xrightarrow[o]{r}\!\!\!\!\gg \lambda x.\, E \mapsto z.\, C \mid E \mid P(t^*) \\
\mathit{EqConstraint} \ni E & ::= & t \stackrel{?}{=} t \mid \mathsf{dataOf}(t, t) \mid E * E \mid \exists x.\, E
\end{array}
$$

Figure 4.6: Syntax of Statix terms and constraints.

The syntax of queries is $t \xrightarrow[o]{r}\!\!\!\!\gg \lambda x.\, E \mapsto z.\, C$. Here, the term t represents a source scope term; r represents a regular expression that determines reachability; o is a partial order that determines visibility; $\lambda x.\, E$ is a *data-well-formedness constraint* which characterizes whether a target scope and its associated data matches the query; z is a set variable that will be bound in C to the result of the query. The syntax used by Rouvoet et al. provides a separate constraint for applying the visibility ordering o. We include this ordering as a part of the query, following the syntax used by the implementation of Statix found in the Spoofax language workbench (Kats and Visser, 2010a).[6]

Also in contrast to Rouvoet et al., we distinguish equality constraints (ranged over by E) from plain constraints (C). This way, data well-formedness predicates of queries ($\lambda x.\, E$) use constraints that can only inspect terms and data, but cannot extend the scope graph. Another difference from Rouvoet et al. is that we define the semantics of predicate constraints. The constraint $P(t^*)$ represents an invocation of a user-specified predicate, such as those from Fig. 4.4. The rules we discuss next are parameterized by a specification $\$$ comprising rules of the form $P(t^*)$.

In the syntax of constraints and terms in Fig. 4.6 and throughout the chapter, we use $\bar{\bullet}$ notation to represent (possibly empty) sequences, and $\bullet^*$ notation to represent their syntax. For example, $P(t^*)$ represents the syntax of a predicate symbol followed by a parenthesized sequence of terms. We use $x; y$ for sequences that can be freely reordered (e.g., $x; y \approx y; x$) and $x \cdot y$ for sequences that cannot (e.g., $x \cdot y \not\approx y \cdot x$). We overload notation and use $x; y$ and $x \cdot y$ to represent sequences both when x is an element and when x is a sequence, and similarly for y.

[6] `https://spoofax.dev/`

$$
\begin{array}{rcl}
 & & h \in \mathit{Hole} \\
\mathit{Configuration} \ni \kappa & ::= & \left\langle \mathscr{G} \,\middle|\, C^* \,\middle|\, U \in (\mathit{Var} \rightharpoonup \mathit{Hole}) \,\middle|\, H \in (\mathit{Hole} \rightharpoonup (s^* \times t)) \right\rangle \\
\mathit{ScopeGraph} \ni \mathscr{G} & ::= & \langle S \subseteq \mathit{Scope},\ R \subseteq (\mathit{Scope} \times \mathit{Label} \times \mathit{Scope}),\ \rho \in (\mathit{Scope} \rightharpoonup \mathit{Term}) \rangle
\end{array}
$$

Figure 4.7: Syntax of Statix configurations and scope graphs.

4.4.2 Operational Semantics of Statix with Hole State Tracking

The operational semantics in Fig. 4.8 also follow Rouvoet et al. (2020). The transition relation uses the configurations whose syntax is given in Fig. 4.7. A configuration $\left\langle \mathscr{G} \,\middle|\, \overline{C} \,\middle|\, U \,\middle|\, H \right\rangle$ comprises:

- $\mathscr{G}$: the currently constructed scope graph.
- $\overline{C}$: the current set of constraints.
- $U \in (\mathit{Var} \rightharpoonup \mathit{Hole})$: associates unification variables with holes. This lets us determine to which hole a given constraint might relate.
- $H \in (\mathit{Hole} \rightharpoonup (s^* \times t))$: maps each hole in the program to its state, consisting of a list of traversed scopes s^* and the term t constructed so far.

A main difference from Rouvoet et al. is that we extended the configuration to track the state of reference synthesis *holes* via the entities U and H. While the rules in Fig. 4.8 never access them, U and H are explicitly propagated by these rules such that substitutions resulting from unification get applied to them. We return to the role of U and H in Section 4.4.3.

Predicate Constraints Rule Op-Pred defines the semantics of *predicate expansion*. To support this rule, the transition judgment in Fig. 4.8 is parameterized by a specification \$. This specification is given by a set of predicate rules where each rule has the shape $P(\bar{t}) \leftarrow C$. Each rule is closed (i.e., $FV(P(\bar{t}) \leftarrow C) = \varnothing$), and we assume that the domain of every predicate rule is disjoint from all other rules; i.e.:

$$\forall P\, \overline{t_1}\, \overline{t_2}\, C_1\, C_2.\ \big(P(\overline{t_1}) \leftarrow C_1\big) \in \$ \wedge \big(P(\overline{t_2}) \leftarrow C_2\big) \in \$ \wedge (\exists \theta.\mathrm{mgu}(\overline{t_1}, \overline{t_2}) = \theta) \Rightarrow \overline{t_1} \equiv \overline{t_2} \wedge C_1 \equiv C_2$$

We also assume that premises that access rules in a specification \$ (e.g., $(P(\bar{t}) \leftarrow C) \in \$$) are automatically α-renamed to avoid variable capture. Rule Op-Pred thus expands a predicate only when there exists a *unique* rule $P(\overline{t_2}) \leftarrow C$ in specification \$ whose head matches the predicate constraint $P(\overline{t_1})$. In case the predicate constraint matches multiple rules in \$, rule Op-Pred does not apply. For example, if P is a predicate symbol, f and g are constructors, x, y are variables, and we have a specification with rules $P(f(x)) \leftarrow C_1$ and $P(g(x)) \leftarrow C_2$,

$\mathbb{S} \vdash \kappa \to \kappa'$ — Configuration κ steps to κ' using specification $\mathbb{S}$

Op-Conj
$$\mathbb{S} \vdash \langle\, \mathcal{G} \mid (C_1 * C_2); \overline{C} \mid U \mid H \,\rangle \to \langle\, \mathcal{G} \mid C_1; C_2; \overline{C} \mid U \mid H \,\rangle$$

Op-Emp
$$\mathbb{S} \vdash \langle\, \mathcal{G} \mid \mathsf{emp}; \overline{C} \mid U \mid H \,\rangle \to \langle\, \mathcal{G} \mid \overline{C} \mid U \mid H \,\rangle$$

Op-Exists
$$\frac{y \text{ is fresh}}{\mathbb{S} \vdash \langle\, \mathcal{G} \mid (\exists x.\, C); \overline{C} \mid U \mid H \,\rangle \to \langle\, \mathcal{G} \mid C[y/x]; \overline{C} \mid U \mid H \,\rangle}$$

Op-Eq
$$\frac{\mathsf{mgu}(t_1, t_2) = \theta}{\mathbb{S} \vdash \langle\, \mathcal{G} \mid (t_1 \stackrel{?}{=} t_2); \overline{C} \mid U \mid H \,\rangle \to \langle\, \mathcal{G} \mid \overline{C} \mid U \mid H \,\rangle\theta}$$

Op-Singleton
$$\mathbb{S} \vdash \langle\, \mathcal{G} \mid \mathsf{single}(t, \{t'\}); \overline{C} \mid U \mid H \,\rangle \to \langle\, \mathcal{G} \mid (t \stackrel{?}{=} t'); \overline{C} \mid U \mid H \,\rangle$$

Op-Forall
$$\mathbb{S} \vdash \langle\, \mathcal{G} \mid (\forall x \text{ in } \zeta.\, C); \overline{C} \mid U \mid H \,\rangle \to \langle\, \mathcal{G} \mid \{C[t/x] \mid t \in \zeta\}; \overline{C} \mid U \mid H \,\rangle$$

Op-New-Scope
$$\frac{s \notin S}{\mathbb{S} \vdash \langle\, \langle S, R, \rho \rangle \mid (\forall x \mapsto t); \overline{C} \mid U \mid H \,\rangle \to \langle\, \langle s; S, R, \rho[s \mapsto t] \rangle \mid \overline{C} \mid U \mid H \,\rangle[s/x]}$$

Op-New-Edge
$$\mathbb{S} \vdash \langle\, \langle S, R, \rho \rangle \mid (s_1 \xrightarrow{\ell} s_2); \overline{C} \mid U \mid H \,\rangle \to \langle\, \langle S, (s_1, \ell, s_2); E, \rho \rangle \mid \overline{C} \mid U \mid H \,\rangle$$

Op-Data
$$\frac{\rho_{\mathcal{G}}(s) = t_2}{\mathbb{S} \vdash \langle\, \mathcal{G} \mid \mathsf{dataOf}(s, t_1); \overline{C} \mid U \mid H \,\rangle \to \langle\, \mathcal{G} \mid (t_1 \stackrel{?}{=} t_2); \overline{C} \mid U \mid H \,\rangle}$$

Op-Query
$$\frac{A = \mathsf{Ans}(\mathcal{G}, s \xrightarrow[o]{r}\!\!\!\!\rightarrow \lambda x.\, E) \qquad \mathsf{guard}(\mathcal{G}, (s \xrightarrow[o]{r}\!\!\!\!\rightarrow \lambda x.\, E \mapsto z.\, C), \overline{C})}{\mathbb{S} \vdash \langle\, \mathcal{G} \mid (s \xrightarrow[o]{r}\!\!\!\!\rightarrow \lambda x.\, E \mapsto z.\, C); \overline{C} \mid U \mid H \,\rangle \to \langle\, \mathcal{G} \mid C[A/z]; \overline{C} \mid U \mid H \,\rangle}$$

Op-Pred
$$\frac{\exists!(P(\overline{t_2}) \leftarrow C) \in \mathbb{S}.\ \exists\theta.\ \mathsf{mgu}(\overline{t_1}, \overline{t_2}) = \theta}{\mathbb{S} \vdash \langle\, \mathcal{G} \mid P(\overline{t_1}); \overline{C} \mid U \mid H \,\rangle \to \langle\, \mathcal{G} \mid C; \overline{C} \mid U \mid H \,\rangle\theta}$$

$\mathcal{G} \vdash E \rightsquigarrow \theta$ — Equality constraint E produces a unifier θ

E-Eq
$$\frac{\mathsf{mgu}(t_1, t_2) = \theta}{\mathcal{G} \vdash t_1 \stackrel{?}{=} t_2 \rightsquigarrow \theta}$$

E-DataOf
$$\frac{\mathsf{mgu}(\rho_{\mathcal{G}}(s), t) = \theta}{\mathcal{G} \vdash \mathsf{dataOf}(s, t) \rightsquigarrow \theta}$$

E-Conj
$$\frac{\mathcal{G} \vdash E_1 \rightsquigarrow \theta_1 \qquad \mathcal{G} \vdash E_2 \rightsquigarrow \theta_2}{\mathcal{G} \vdash E_1 * E_2 \rightsquigarrow \theta_1\theta_2}$$

E-Exists
$$\frac{y \text{ is fresh} \qquad \mathcal{G} \vdash E[y/x] \rightsquigarrow \theta}{\mathcal{G} \vdash \exists x.\, E \rightsquigarrow \theta}$$

Figure 4.8: Operational semantics of constraints in Statix.

then the Op-Pred rule does not apply to the predicate constraint $P(y)$ because y can be instantiated to both $f(x)$ and $g(x)$.

The substitution yielded by mgu in the premise of Op-Pred is applied to the entire configuration after unfolding a predicate. Here and in the rest of the chapter, mgu is the partial function computing the most general unifier (i.e., a substitution). We use $\bot$ to denote failure in partial functions. We use $\theta, \theta_1, \theta_{result}, \ldots$ to range over substitutions of variables by terms ($Var \rightharpoonup Term$) or set variables by set terms ($SetVar \rightharpoonup SetTerm$). The type of substitution will be clear from the context. The substitution functions for constraints, terms, and scope graphs are standard and elided for brevity, except for the reference entity U, which we describe in Section 4.4.3.

Logic Constraints The other rules in Fig. 4.8 are directly adapted from Rouvoet et al. (2020). Rule Op-Conj splits a separating conjunction constraint into two constraints. Op-Emp dispatches the vacuously satisfiable constraint emp. Op-Exists unpacks an existentially quantified constraint by choosing a fresh variable name, which may get unified using, for example, Op-Eq.

Set Constraints Queries yield *sets* of results, so the rules in Fig. 4.8 include two dedicated constraints for matching on sets. The semantics of $\mathsf{single}(t_1, t_2)$ is given in rule Op-Singleton, which asserts that t_2 must be a singleton set $\{t'\}$, such that t_1 unifies with t'. The semantics of constraint $\forall x \ \mathsf{in} \ \underline{t}.\ C$ is given in rule Op-Forall. The rule asserts that $\underline{t}$ must be some set literal ζ; i.e., a union of singleton sets. The rule expands $\forall x \ \mathsf{in} \ \zeta.\ C$ into as many constraints as ζ has singleton sets, in each case substituting x for the singleton set inhabitant.

Scope Graph Constraints The rules Op-New-Scope and Op-New-Edge create new scopes and edges in the scope graph, respectively. Rule Op-Data asserts that a constraint $\mathsf{dataOf}(t_1, t_2)$ can be solved when t_1 is a scope s, and t_2 unifies with the term associated with scope s. Rule Op-Query has two premises. The function Ans returns the set of all paths that match the query parameters (see Section 4.2.1). The guard predicate ensures that the query is *guarded* in the sense that the constraints $C; \overline{C}$ do not add a new edge to the scope graph that would cause the query to yield a different answer. Both Ans and guard are discussed in detail by Rouvoet et al. (2020, §3.1 and §5.3).

4.4.3 Operational Semantics of Reference Synthesis

The usual operational semantics of Statix in Fig. 4.8 is conservative about solving query and predicate constraints. As discussed before, Op-Pred solves a predicate constraint only

when there is exactly one possible expansion, otherwise it is *stuck*. Similarly, Op-Query only solves a query constraint when the source scope of the query is ground (i.e., it is a scope rather than a variable), and the guard premise holds.

As we can observe from the example discussed in Section 4.3, speculatively solving predicate and query constraints allows the Statix solver to *infer* what syntax of valid references to substitute for each hole of a program. To admit such inference, the rules in Fig. 4.9 let us *speculatively* expand *predicate* and *query constraints* in states that would otherwise be stuck. We achieve this by introducing two new relations: $\rightarrowtail$ performs a speculative expansion step, and $\twoheadrightarrow$ relates sets of potentially speculatively expanded configurations.

The $\twoheadrightarrow$ Relation As long as the regular Statix constraint-solving rules can make progress, the Op-Solve rule applies. Once the solver gets stuck, Op-Expand applies. The set comprehension in the bottom premise of the rule lets us speculatively solve stuck query constraints and expand predicates. Configurations for which neither Op-Solve nor Op-Expand apply are truly stuck, and will be pruned by the set comprehension premise of Op-Expand.

Speculative Predicate Expansion The rule Op-Expand-Pred augments the plain Statix constraint-solving rules from Fig. 4.8 to support selecting an *arbitrary* rule for expanding a predicate constraint.

Speculative Query Expansion The rule Op-Expand-Query augments the plain Statix constraint-solving rules from Fig. 4.8 with support for solving a stuck query constraint, by synthesizing a path from a (possibly unknown) source scope to the current target scope of the related hole. The rule assumes that we are resolving a reference given by a composite path, and attempts to "prepend" a step to the composite path. Intuitively, if we think of composite paths as (qualified) references, this corresponds to attempting to prepend a qualifier.

Op-Expand-Query uses the U and H components of the state to determine which hole it is expanding. For each free variable in the program, U tracks to which hole it is related. For this reason, its substitution function $U[t/x]$ has a guard that checks that each free variable in t is either not related to a hole, or is related to the same hole as x. As shown previously in Fig. 4.7, the state of a hole $H(h)$ is given by a pair $(\bar{s}, t)$. Here, t is a term representing the inferred syntax for the reference, while $\bar{s}$ represents a non-empty sequence of *query-connected scopes* that form its composite path.

Definition 4.4.1 (Query-Connected Scopes)**.** For a given scope graph $\mathcal{G}$, two scopes s_1 and s_2 in $\mathcal{G}$ are *query-connected* by a query $q = s_1 \xrightarrow[o]{r} \!\!\!\!\!\rightarrow \lambda x.\, E$, which we denote $s_1 \xrightarrow{q} s_2$, when there exists an p such that $p \in \mathsf{Ans}(\mathcal{G}, s_1 \xrightarrow[o]{r} \!\!\!\!\!\rightarrow \lambda x.\, E)$ where either $s_2 = \mathsf{tgt}(p)$ or $s_2 \in \rho_{\mathcal{G}}(\mathsf{tgt}(p))$.

This uses the notation $s_2 \in \rho_{\mathscr{G}}(\mathrm{tgt}(p))$ to mean that s_2 is a syntactic subterm of the data associated with $\mathrm{tgt}(p)$. Using the definition, we can define composite paths that model references.

Definition 4.4.2 (Composite Path)**.** For a given scope graph $\mathscr{G}$, a sequence of scopes $s_0 \dots s_n$, and a sequence of queries $q_i \in \overline{q}$, a composite path in the scope graph is given by a series of query-connected scopes; i.e.: $s_0 \xrightarrow{q_1} s_1 \cdots \xrightarrow{q_n} s_n$

The first premise of rule Op-Expand-Query asserts that the data well-formedness predicate $(\lambda y.\, E)$ has a variable occurring in it, which is relevant for a hole h. The head scope s_t of the composite path component in the state of h represents the scope that we need to connect to, in order to prepend a step to a query-connected path. The remaining premises assert that we choose a source scope s' and a target scope s'' that is connected to s_t, such that the query resolves from s' to s''.

Accepting States The $\mathrm{Accept}(\overline{C}, H)$ premise of Op-Expand holds iff (1) $\overline{C}$ is empty (all constraints are solved), and (2) the state of each hole in H has a composite path component of length > 1 (we have constructed a composite path for each hole).

4.4.4 Building the Synthesize Function

Now, we have all the pieces to build the synthesize function:

$$\frac{\begin{array}{c} D = \{x \mapsto d \mid (\!|{\to}d|\!) \in \mathscr{P}_{\text{locked}}, x \text{ fresh}\} \\ U = \{x \mapsto h \mid x \in \mathrm{dom}(D), h \text{ fresh}\} \qquad \theta_1 = \{(\!|{\to}d|\!) \mapsto x \mid D(x) = d\} \\ \mathscr{P}_0 = \mathscr{P}_{\text{locked}}\theta_1 \qquad \langle\, \varnothing \mid P_0(\mathscr{P}_0) \mid U \mid \varnothing \,\rangle \to^* \langle\, \mathscr{G}_0 \mid \overline{C} \mid U \mid \varnothing \,\rangle \\ H = \{h \mapsto (s_d, x) \mid U(x) = h, D(x) \in \rho_{\mathscr{G}_0}(s_d)\} \qquad \langle\, \mathscr{G}_0 \mid \overline{C} \mid U \mid H \,\rangle \twoheadrightarrow^* \overline{\kappa} \\ \langle\, \mathscr{G} \mid \overline{C}' \mid U' \mid H' \,\rangle \in \overline{\kappa} \qquad \mathrm{Accept}(\overline{C}', H') \qquad \theta_{result} = \{x \mapsto t \mid H'(U(x)) = (s_t, t)\} \end{array}}{\mathrm{synthesize}(\mathscr{P}_{\text{locked}}) = \mathscr{P}_0\theta_{result}}$$

We first create a fresh unification variable and hole for each locked reference in the program (D and U, respectively), and replace each locked reference by a fresh unification variable in the program (θ_1). Then, we solve the initial constraint (P_0) on the program with holes. From the partial scope graph in the result state, we initialize the hole state H for each hole. Then, we synthesize the references and extract an accepted state. From this state, we build a substitution θ_{result} that substitutes each hole variable with the synthesized reference term.

$\$ \vdash \overline{\kappa} \twoheadrightarrow \overline{\kappa'}$ — Configurations $\overline{\kappa}$ step to $\overline{\kappa'}$ using specification $\$$

$$\textsc{Op-Solve}\ \frac{\$ \vdash \kappa \to \kappa'}{\$ \vdash \kappa; \overline{\kappa} \twoheadrightarrow \kappa'; \overline{\kappa}} \qquad \textsc{Op-Expand}\ \frac{\$ \vdash \left\langle \mathcal{G} \mid \overline{C} \mid U \mid H \right\rangle \not\to \quad \neg\mathsf{Accept}(\overline{C}, H) \quad \overline{\kappa'} = \left\{ \left\langle \mathcal{G} \mid \overline{C} \mid U \mid H' \right\rangle \theta \mid \$ \vdash \left\langle \mathcal{G} \mid \overline{C} \mid U \mid H \right\rangle \rightarrowtail \theta, H' \right\}}{\$ \vdash \left\langle \mathcal{G} \mid \overline{C} \mid U \mid H \right\rangle; \overline{\kappa} \twoheadrightarrow \overline{\kappa'}; \overline{\kappa}}$$

$\$ \vdash \kappa \rightarrowtail \theta, H$ — For κ, synthesize substitution θ and hole state H, using specification $\$$

$$\textsc{Op-Expand-Pred}\ \frac{(P(\overline{t_2}) \leftarrow C) \in \$ \qquad \mathrm{mgu}(\overline{t_1}, \overline{t_2}) = \theta}{\$ \vdash \left\langle \mathcal{G} \mid P(\overline{t_1}); \overline{C} \mid U \mid H \right\rangle \rightarrowtail \theta, H}$$

$$\textsc{Op-Expand-Query}\ \frac{\begin{array}{c} h \in \{h \mid x \in FV(\lambda y.\, E), h = U(x)\} \qquad H(h) = \left(s_t \cdot \overline{s'_t}, t'\right) \\ s', s'' \in S_{\mathcal{G}} \qquad \mathrm{mgu}(t, s') = \theta_1 \qquad (s_t = s'' \vee s_t \in \rho_{\mathcal{G}}(s'')) \\ \mathcal{G} \vdash E[s''/y]\theta_1 \rightsquigarrow \theta_2 \qquad \exists p \in \mathsf{Ans}(\mathcal{G}\theta_1\theta_2, s' \xrightarrow[o]{r}\!\!\!\!\twoheadrightarrow \lambda y.\, E\theta_1\theta_2).\ \mathrm{tgt}(p) = s'' \\ \mathrm{guard}(\mathcal{G}\theta_1\theta_2, (s' \xrightarrow[o]{r}\!\!\!\!\twoheadrightarrow \lambda x.\, E \mapsto z.\, C)\theta_1\theta_2, \overline{C}) \end{array}}{\$ \vdash \left\langle \mathcal{G} \mid t \xrightarrow[o]{r}\!\!\!\!\twoheadrightarrow \lambda y.\, E \mapsto z.\, C; \overline{C} \mid U \mid H \right\rangle \rightarrowtail \theta_1\theta_2, H\left[h \mapsto \left(s' \cdot s_t \cdot \overline{s'_t}, t'\right)\right]}$$

Auxiliary functions

$$U[t/x] = \begin{cases} U & \text{if } x \notin \mathrm{dom}(U) \\ U[\{y \mapsto U(x) \mid y \in FV(t)\}] & \text{if } x \in \mathrm{dom}(U) \text{ and } \forall y \in FV(t), h'.\, U(y) = h' \Rightarrow h' = U(x) \\ \bot & \text{otherwise} \end{cases}$$

Figure 4.9: Operational semantics of reference synthesis.

4.4.5 Properties

In this section, we discuss some properties of our operational semantics: soundness, completeness, and confluence; and we discuss liveness.

Soundness First, we consider the *soundness* of our approach. Soundness consists of two components: (1) the resulting solutions are well-typed; and (2) each solution corresponds to a composite path in the scope graph to the target declaration.

Theorem 4.4.3 (Soundness 1)**.** *Programs with synthesized references are well-typed.*

Theorem 4.4.4 (Soundness 2)**.** *Every synthesized reference corresponds to a composite path (Theorem 4.4.2) to the target it was initially locked to.*

Theorems 4.4.3 and 4.4.4 have formal definitions and proofs in Appendices B.1.2 and B.1.3, respectively.

Completeness Ideally, we would conjecture *completeness* as well. However, completeness is hard to define, as it must rely on a generic (language-independent) definition of a reference. For the purpose of this chapter, we consider a well-formed reference in scope s_0 to declaration d to be a composite path (Theorem 4.4.2) through scope graph $\mathscr{G}$, from s_0 to s_d where $d \in \rho_{\mathscr{G}}(s_d)$ (i.e., s_d is associated with declaration d). However, this definition is an over-approximation, as queries could be connected 'by accident'. For example, a Java expression `a.m(b)` where `a` is an instance of the class in which this expression occurs could be considered a reference `a.b` by our definition, as the target of the query for `a` would match the source of the query of `b`. Thus, our definition is not suitable to state a completeness theorem. In Section 4.6.3 we show experimental evidence that our approach is practically complete.

Confluence We also consider the property of *confluence*: if two different expansions are possible, eventually, the final state sequence will be equivalent.

Theorem 4.4.5 (Confluence). *If* $\overline{\kappa} \twoheadrightarrow \overline{\kappa}_1$ *and* $\overline{\kappa} \twoheadrightarrow \overline{\kappa}_2$, *then* $\exists \overline{\kappa'}.\ \overline{\kappa}_1 \twoheadrightarrow^* \overline{\kappa'} \wedge \overline{\kappa}_2 \twoheadrightarrow^* \overline{\kappa'}$

Proof. This is a proof by case analysis on the expanded state:

- If different states were expanded in $\overline{\kappa}_1$ and $\overline{\kappa}_2$, the step made to obtain $\overline{\kappa}_1$ can be applied on $\overline{\kappa}_2$ as well, and vice versa. This yields equivalent states again.
- If the same state was expanded in $\overline{\kappa}_1$ and $\overline{\kappa}_2$, either one of the following is true:
  - An Op-Solve-step was made in both cases. In this case, confluence holds by virtue of $\rightarrow$ being confluent (Rouvoet et al., 2020, Theorem 4.5).
  - An Op-Expand-step was made in both cases. As this rule ranges over all possible expansions, $\overline{\kappa}_1 = \overline{\kappa}_2$, so confluence trivially holds.

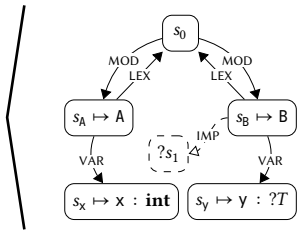
  The same state cannot be expanded with both Op-Solve and Op-Expand, as the first premise of Op-Expand requires the state to be stuck (i.e., Op-Solve does not apply). □

This confluence result is especially important for our next section, as we exploit this property to design heuristics that reduce the huge search space of possible expansions.

Liveness Finally, although not a property of the operational semantics, we discuss *liveness* here as well. Liveness consists of two components: (1) the synthesis finds each solution in finite time, and (2) when no (new) solution is available, the synthesis terminates. Property (1) can be guaranteed by scheduling the expansion steps 'fairly'; i.e., in a breadth-first manner. Property (2) requires special care for predicates that are (potentially) infinitely expanding without yielding a solution. We return to this in Section 4.5.5.

```
1 mod A {
2   var x = 1
3 }
4 mod B {
5   import (→A) ::*
6   var y = (→x)
7 }
```

$$H = \begin{cases} ?A_1 & \mapsto h_A \\ ?x & \mapsto h_x \\ ?A_2 & \mapsto h_x \end{cases} \qquad U = \begin{cases} h_A & \mapsto (s_{\mathsf{A}}, ?A_1) \\ h_x & \mapsto (s_{\mathsf{x}}, ?A_2.?x) \end{cases}$$



$$\begin{array}{rll} \mathsf{scopeOfMod}(s_{\mathsf{B}}, ?A_1, ?s_1) & & (1) \\ s_{\mathsf{B}} \xrightarrow{\mathsf{IMP}} ?s_1 & & (2) \\ \mathsf{scopeOfMod}(s_{\mathsf{B}}, ?A_2, ?s_2) & & (3) \\ ?s_2 \xrightarrow{\mathsf{VAR}} \mathsf{isVar}_{?x} \mapsto \{(_, ?x : ?T)\} & & (4) \end{array} \quad \Big| \; U \; \Big| \; H \; \Big\rangle$$

Figure 4.10: Example program (top-left) and intermediate synthesis state (below and top-right).

4.5 Heuristics

The operational semantics is highly non-deterministic. A direct, naive implementation would perform duplicate and unnecessary work. To reduce work, our implementation of reference synthesis uses heuristics to guide the search. These heuristics have three goals: (i) to guide the search toward results, (ii) to avoid duplicate work, and (iii) to cut search branches early that do not lead to results. For all these heuristics, we argue that they do not yield solutions that are not derivable from the operational semantics (soundness), and preserve all derivable solutions as well (completeness).

In this section, we discuss these heuristics, using the example in Fig. 4.10. Performing reference synthesis on the program on the top-left eventually reaches the state shown in the figure. The first two constraints are obtained from expanding rule D-ImportOk on the initial hole on line 5 (h_A), represented by the variable $?A_1$. Resolving this reference yields a scope, currently represented by the variable $?s_1$. Once this scope is resolved, an incoming edge from s_{B} can be created. Until then, this edge is not present in the scope graph (hence it is indicated with a dashed line). The other two constraints correspond to the hole on line 6 (h_x), which is expanded to a reference with a single qualifier $?A_2.?x$. The qualifier should resolve to a scope $?s_2$, in which afterward, a query resolving to the target scope s_{x} will be performed.

4.5.1 Selecting Constraints

As discussed in Section 4.4.5, our system is *confluent*. For that reason, we can choose one single order in which to expand constraints, instead of trying all possible orders. We choose the constraint to expand according to the following criteria (considered in order): (i) prefer queries over predicate constraints; (ii) prefer predicate constraints that can lead to queries over those that cannot; and (iii) prefer older constraints over newer constraints. The rationale behind this order is that we try to expand queries as soon as possible. Expanding queries typically blows up the search space less than expanding predicates, and queries often reach terminal states (ground references). Using age as a tiebreaker, we try to explore the remaining state space in a breadth-first manner, to ensure we reach all possible references. In our example, this implies that we first expand the query constraint (4), inferring x to be the value of $?x$, and s_{A} to be the target $?s_2$ of constraint (3). As we eventually expand all constraints, we preserve soundness and completeness.

4.5.2 Expanding Queries

When expanding a query, a source and target scope must be chosen (s' and s'' in Op-Expand-Query in Fig. 4.9). Instead of trying the query with all possible combinations of source and target scopes, we can be smart about the source and target scopes we pick. First, we choose s'', based on the premise that it should be equal to the current target scope, s_t, or contain it. Then, we infer a unifier from the query, $\mathcal{G} \vdash E[s''/y] \rightsquigarrow \theta$. If no such unifier exists, we stop the search for this branch. Finally, we traverse the scope graph backward, starting at s''. We use the inverted regular expression (e.g., $\mathsf{inv}(\mathsf{LEX}^*\ \mathsf{VAR}) = \mathsf{VAR}\ \mathsf{LEX}^*$) to guide the graph traversal to only find source scopes that have a valid path to the target scope. This approach only over-approximates when there is another declaration, reachable from s', that shadows s''. It is sound, as eventually we check all the premises of the rule. It is also complete, as all other choices for s'' do not satisfy the premise that relates it to s_t, and all other choices for s' do not yield valid query answers.

In our example, when expanding the query constraint (4), the only reasonable choice for s'' is s_{x}. Then, solving E yields $\{?x \mapsto \mathsf{x}\}$. Next, following the inverted regular expression (VAR) traversing the edge $s_{\mathsf{A}} \xrightarrow{\text{VAR}} s_{\mathsf{x}}$ backward, we choose s' to be s_{A}. This instantiation is a valid query expansion.

4.5.3 Expanding Predicates

When selecting a predicate (Section 4.5.1) we fork the computation and try each possible expansion of a predicate to a rule concurrently. We prioritize expansions that lead to a query, as those might converge to a result quicker. Additionally, we prioritize rules that

have fewer free variables, as those add less freedom to the problem; i.e., tell us more about the final solution. As we only prioritize certain rules over others, but never discard any, we will eventually try all rules. Therefore, this does not affect completeness. In our example, this implies that we prioritize expanding the scopeOfMod constraints using rule S-Mod before rule S-QMod.

4.5.4 Isolating Holes

The schemes in Section 4.5.1 and Section 4.5.3 reduce the search space significantly, but may skew the search to holes with less complicated solutions. Therefore, when starting the computation, we fork the state for each hole, which we call the *focus hole* of that search branch. We only expand constraints that are related to this focus hole, ensuring we make progress on this hole specifically.

However, this approach is incomplete, as sometimes the solution of one hole can only be computed after another hole is solved. For example, for the program in Fig. 4.10, `x` is a valid solution for h_x (although in a different branch than the state presented there). However, this can only be computed when the solution for h_A is known, as it requires the edge $s_{\mathsf{B}} \xrightarrow{\text{IMP}} s_{\mathsf{A}}$ to be present in the scope graph. We need a way to ensure such 'composite' solutions are computed as well.

We ensure this as follows. Suppose query expansion (Section 4.5.2) traverses an l-labeled edge. When there is a scope s such that future steps *might* create an l-labeled edge in s (i.e., $\overline{C} \hookrightarrow (s, l)$; see Rouvoet et al. (2020, §5.3)), we might miss edges. In this case, we find the constraint that is responsible for the missing edge, and all constraints that (transitively) share a variable with this constraint. Some of these constraints may correspond to a different hole. Then, we peek at that hole for solutions and try to insert those solutions in our current state. Solving this state should create the missing edge. Next, we can resume our backward traversal. Since we over-approximate the missing edges ($\hookrightarrow$ over-approximates, and the future edges may also have different target scopes), we preserve completeness. In addition, we do not change the traversal itself, so we also preserve soundness.

In the example, we detect that constraint (2) is responsible for the missing edge. As this constraint shares a unification variable with constraint (1), we conclude it is related to h_A. Thus, we insert a solution for h_A (e.g., `A`) in the state, after which we can find the solution `x` for h_x.

4.5.5 Recursive Qualifiers

Another heuristic concerns *recursive qualifiers*. We consider a reference recursive if multiple qualifiers in the reference resolve to the same declaration. For example, consider a Java class

`A` with field `int x` and a field `A a`. In this case, a hole referring to `x` has an infinite number of possible solutions: `x`, `a.x`, `a.a.x`, etc. In these cases, each `a` refers to the same declaration.

To explain how we optimize synthesizing these references, we consider two of the states the synthesis of `a.a.x` goes through. After some steps, the synthesis reaches this state:

$$\kappa_1 = \langle \mathcal{G} \mid \mathsf{resolveQVar}(s_\mathsf{A}, ?q_1, ?s_1) \mid \{?q_1 \mapsto h, \ldots\} \mid \{h \mapsto (s_\mathsf{A} \cdot s_\mathsf{x}, ?q_1.\mathsf{x}), \ldots\} \rangle$$

Some forks of this branch will arrive at `a.x` as a solution for this hole. However, on the path to synthesizing `a.a.x`, the following state will also be reached:

$$\kappa_2 = \langle \mathcal{G} \mid \mathsf{resolveQVar}(s_\mathsf{A}, ?q_2, ?s_2) \mid \{?q_2 \mapsto h, \ldots\} \mid \{h \mapsto (s_\mathsf{A} \cdot s_\mathsf{A} \cdot s_\mathsf{x}, ?q_2.\mathsf{a.x}), \ldots\} \rangle$$

These states are very similar: they share the same scope graph, have α-equivalent constraints, and the same target scope in the hole state of h. For that reason, we can conclude that *each solution* for $?q_1$ is *also a solution* for $?q_2$, as any derivation starting in κ_1, is also valid in κ_2. Therefore, we can avoid synthesizing the same solution multiple times by reusing solutions for $?q_1$ when synthesizing $?q_2$. As the traces are equivalent, we preserve soundness and completeness.

Algorithmically, we achieve this in the following three steps: (1) We detect pairs of solver states where (a) the constraints are equivalent up to α-renaming, (b) the hole's term in the recursive solver state is an instantiation of the term in the base state, and (c) the hole has the same target scope. For these states, we stop further synthesis on the recursive state. (2) For each ground solution for the variable in the base state, we emit a solution derived from the recursive state. (3) Repeat from step 2, using this new solution.

In the example above, this detects that κ_2 is a recursive state with respect to κ_1. Thus, solutions from other branches rooted in κ_1 (such as `a.x`) are reused in the variable $?q_2$ in κ_2, yielding `a.a.x`.

This expansion is interleaved with executing other search branches, in order to guarantee liveness. In the case where no base solutions exist, this immediately terminates a search branch that would otherwise run infinitely without returning results. This ensures that we are terminating on recursive instances that match this pattern. When we assume the predicates that model references do not generate new scopes, only a finite number of recursive instances can be generated (because there exists a finite number of scopes in the scope graph, and a finite number of rules in a specification; hence only a finite number of states that are not equivalent according to the definition in step 1 above exist.). That implies that we can only have non-termination when the recursive reference predicate generates fresh scopes, which typical specifications do not do.

	Java	ChocoPy	FGJ
Included	2528 (62%)	196 (64%)	49 (94%)
Negative	0 (0%)	55 (18%)	0 (0%)
Java 8 incompat.	1076 (26%)	N/A	N/A
Spec incompat.	197 (5%)	10 (3%)	0 (0%)
No references	304 (7%)	47 (15%)	3 (6%)
Total	4105 (100%)	308 (100%)	52 (100%)

Figure 4.11: Test selection.

The next section evaluates our implementation, which is based on these heuristics, providing evidence that we indeed preserve soundness and completeness.

4.6 Experimental Evaluation

In evaluating reference synthesis, we focus on three key criteria: (1) ensuring that synthesized references are valid according to the static-semantic specification of the language and resolve to the intended declaration (*soundness*), (2) verify the ability to find a valid reference if it exists (*completeness*), and (3) assess the efficiency of the synthesis process (*performance*). In this section, we discuss how we evaluated these aspects of reference synthesis.

4.6.1 Languages

We evaluated our synthesize function on three larger programming languages: Java, ChocoPy, and Featherweight Generic Java (FGJ).

1. *Java* is a mainstream language with sophisticated name-binding features. We evaluated our approach using an existing Statix specification of Java 8 from the artifact (van Antwerpen and Visser, 2021a) associated with the work of van Antwerpen and Visser (2021b). This specification of Java, unfortunately, does not support generics or method references. We derived test cases from Java files used to validate the implementation of refactorings in the JetBrains IntelliJ IDEA development environment.[7]

2. *ChocoPy* (Padhye, Sen, and Hilfinger, 2019) is a statically-typed dialect of Python. We used an existing Statix specification and ChocoPy files for our evaluation.

3. *Featherweight Generic Java* (FGJ; Igarashi, Pierce, and Wadler, 2001) is a functional Java core language with full generics support. We used the Statix specification from the artifact (van Antwerpen, Bach, et al., 2018a) of van Antwerpen, Bach, et al. (2018b).

[7] https://github.com/JetBrains/intellij-community/tree/idea/233.14808.21/java/java-tests/testData/refactoring

	Java	ChocoPy	FGJ
Success	2382 (94%)	155 (79%)	38 (78%)
Timeout	146 (6%)	41 (21%)	11 (22%)
Failure	0 (0%)	0 (0%)	0 (0%)
Total	2528 (100%)	196 (100%)	49 (100%)

Figure 4.12: Test outcomes.

4.6.2 Method

For each language, we used the existing Statix semantic specifications *without modification*, as our synthesize function works directly with these specifications. We collected a set of test cases: single-file programs where we locked variable names, type names, and qualified member names that occur in them. As a result, many test cases contain multiple locked references. Each locked reference only has the target declaration as a parameter. The original syntax of the reference is erased. We excluded negative test cases (tests that validated that the specification or implementation gives an error on incorrect programs), those that are incompatible with our Statix specification or Java 8 (in the case of Java tests), and those that had no references to lock. The resulting selection is shown in Fig. 4.11. Our reference synthesis algorithm was then applied to propose concrete references until the original reference in the program was recovered. We set a timeout of 60 s per test case and ran the evaluation on a MacBook Pro 2019 with a 2.4 GHz Intel Core i9 processor and 64 GB memory.

4.6.3 Results

A summary of the results of the evaluation is shown in Fig. 4.12. The results confirm the *soundness* of our reference synthesis algorithm. Substituting the synthesized references back into the original programs only produced well-typed programs (Theorem 4.4.3).

Our evaluation demonstrates a high level of *completeness*, as it successfully synthesized each reference encountered in our test cases, including non-trivial references such as Java's `A.super.x` and references with ambiguous qualifiers (Gosling et al., 2015, §6.5.2). Nevertheless, formally proving *completeness* of the algorithm is challenging (Section 4.4.5).

While *performance* was not the main focus of our approach, we also measured the time taken to synthesize each reference, shown as a violin plot in Fig. 4.13. The plot illustrates the distribution of the synthesis time per hole as a density curve, where for all three languages, most results lie between 10 and 200 milliseconds. A small box plot is depicted inside each density curve, highlighting the median, range of the central 50% results, and range of the remaining data. For the majority of locked references the algorithm proposed a solution within one second. In around 7% of the test cases, reference synthesis failed to find a solution for all locked references within the 60-second timeout. This typically occurred when

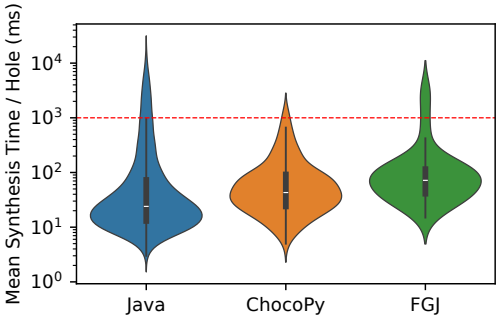


Figure 4.13: Logarithmic plot of the time spent synthesizing references per hole in each of 2575 successful test cases. The dashed line marks 1 second.

references were strongly interdependent, as reference synthesis will exhaustively explore all combinations of solutions for the dependent and the dependency (see Section 4.5.4).

4.6.4 Threats to Validity

To mitigate bias in the references we lock, or the alternatives we try, we make two conservative assumptions that are not generally true for a refactoring: (1) *all* references in the program must be locked; and (2) the original reference syntax is unknown. These assumptions make our test cases more challenging and may result in a worse performance than would be expected in practical scenarios.

Instead of locking all references in the program, a typical refactoring would only need to lock a subset of those references, namely those that could get changed due to the program transformation. Consequently, the likelihood of interdependent locked references would be lower. Furthermore, most existing references are likely to remain valid despite the transformation. Therefore, reference synthesis could prioritize verifying that the existing reference syntax still resolves to the intended declaration before synthesizing a new concrete reference. As a result, only a handful of references need to be synthesized.

Another potential threat to validity is that our approach is parameterized by the language's static semantics written in Statix. As shown by Rouvoet et al. (2020), van Antwerpen, Bach, et al. (2018b), and Zwaan and Bach (2024), Statix can express many interesting name binding and type system concepts, but it is yet unclear what language concepts Statix cannot express. The features of Statix that we heavily rely on are the solver interface, the presence of predicate constraints and query constraints, and conservative scheduling. Given that we do not expect significant changes in any of these, we expect that our reference synthesis algorithm will work without fundamental modifications being required by possible future extensions to Statix.

4.7 Related Work

Refactoring as a discipline and subject goes back to the pioneering works by Griswold (1992) and Opdyke (1992). Since then, refactoring has become a well-established field that has grown exponentially. Steimann (2018) notes that the growth has been so large that a recent attempt to update the Mens and Tourwé's survey (2004) was abandoned as there were too many works to be considered. Consequently, we focus our discussion on prior work most relevant to reference synthesis.

A distinguished line of work on implementing refactorings is the work of Li, Thompson et al. on refactoring tools for Haskell (2003, 2005) and Erlang (2006), which provide support for scripting general-purpose code transformations with possibilities to specify pre- and post-conditions that ensure the transformation preserves certain properties explicitly. This support is realized via name-binding APIs implemented from scratch for each individual language.

As discussed in the introduction, our work builds upon previous work by Ekman, Schäfer, and Verbaere (2008), Schäfer, Ekman, and de Moor (2008), Schafer et al. (2012), and Schaefer and de Moor (2010), who introduced the idea of tracking name-binding dependencies by "locking" references to declarations and then synthesizing concrete references. Schäfer and de Moor applied this concept to synthesizing references for Java but deemed their reference synthesis algorithm to be beyond the scope of their chapter (Schaefer and de Moor, 2010, §2). Our reference synthesis algorithm is applicable to any language whose typing rules are defined using Statix (van Antwerpen, Bach, et al., 2018b).

In other closely related previous work, de Jonge and Visser (2012) describe a language-generic API for name-binding preserving refactorings. Their approach is inspired by the work of Ekman et al. (2008) on JastAdd (Ekman and Hedin, 2007), which they generalize by giving operations for querying name-binding information and requalifying names. They demonstrate that this approach could be applied to multiple languages, including Stratego and a subset of Java (Visser, 2001; Visser, Benaissa, and Tolmach, 1998). The main difference between their generic API and our work is that de Jonge and Visser require the requalification function to be explicitly provided. Our work provides this function.

Tip (2007) demonstrates that the problem of checking that a refactoring preserves well-typedness in a language like Java can be expressed as a *type constraint problem* (Palsberg and Schwartzbach, 1994). His type constraint framework supports a wide range of refactorings for a large subset of Java. While his work focuses on Java-specific constraints, we address the broader problem of guaranteeing that references resolve to the same declaration for *any* programming language whose semantics of name resolution is defined using scope graphs, with Java serving as one of the case studies.

Steimann (2018) generalizes the work of Tip to also consider binding preservation and provides a more general and language-independent foundation for constraint-based refactoring. Although this foundation could, in principle, support refactorings in terms of name-binding constraints that would guarantee binding preservation in any language, the question of how to provide a language-parametric semantics for such constraints is left open. Our language-parametric algorithm might provide an answer to this question.

An important aspect of refactorings that move code is to avoid accidental name capture. A common approach to avoiding name capture is *renaming* (following, e.g., the Barendregt convention (1984)). While our reference synthesis algorithm currently focuses on synthesizing (qualified) references, it does not produce suggestions where name capture could be avoided by renaming declarations. The language-parametric Name-Fix algorithm due to Erdweg, van der Storm, and Dai (2014) provides an interesting solution to this problem.

Despite serving a very different purpose, in Chapter 3 we defined a language-parametric code completion algorithm that also relies on the Statix specification. Like this approach, we inserted a free unification variable in a placeholder position, and partially type-checked the program. By inspecting the stuck constraints, we found and proposed type-sound suggestions for code completion. That work, together with this, shows that having a declarative but executable specification of the static semantics is essential to deriving sound, language-parametric editor services.

4.8 Conclusion

We have presented a novel approach to reference synthesis that is *automatic*, *language-parametric*, and *sound* — generating only well-typed references to the intended declarations. This approach is applicable to any language whose static semantics is defined using typing rules in Statix (van Antwerpen, Bach, et al., 2018b). It works out-of-the-box for such languages, is *sound* by construction, and uses non-deterministic search to provide a high degree of *completeness*. Our evaluation demonstrates that our algorithm works on practical examples, but also reveals that our generic approach comes with a high performance cost, which we intend to explore in future work.

4.9 Future Work

To improve the performance of our reference synthesis algorithm, only references in the program that may be affected by the refactoring should be locked. However, it remains unclear how this set can be determined precisely. Even for a relatively simple Rename refactoring, the set of affected references is not limited to references that resolved to the

renamed declaration. References that share a name with either the original or the new name of the declaration can also be affected, as the renaming can cause them to no longer resolve, or to resolve to an unintended declaration due to shadowing. We hypothesize that, for Rename refactorings, the affected references can be approximated by looking at their names and their resolution paths before and after renaming. In particular, a reference may be affected if its resolution paths, when shadowing is ignored, include a path that reaches the declaration before or after renaming. This still yields a conservative set, but excludes references in other parts of the program that are definitely unaffected, despite sharing a name with the original or new name of the declaration.

Another improvement that could benefit both performance and usability is to take the original (qualified) identifier of the reference into account. By trying the original identifier first, the algorithm could quickly determine whether any reference synthesis is necessary at all. If it is, the algorithm could use the syntax of the original identifier as a starting point and explore modifications to it, instead of synthesizing an identifier from scratch. Besides the performance improvement, this would have the added benefit that synthesized references change in a more predictable way. For example, the current implementation allows a qualified identifier to lose all of its qualifiers if the resulting reference still resolves correctly. An improved implementation could instead retain the qualifiers as long as the reference resolves correctly, thereby making the changes to the program smaller and more predictable.

Furthermore, the current approach searches for suitable identifiers in a non-deterministic way. As a result, the same reference may be synthesized differently across different runs of the algorithm. To make the results stable while also improving usability, we propose introducing an ordering function that specifies whether one synthesized identifier is preferred over another. This function could, for example, be implemented to prefer identifiers that are more similar to the original identifier, have fewer qualifiers, or are more explicit. The ordering could also be used to guide the search toward more preferred identifiers, allowing the search to stop once a preferred identifier has been found.

Finally, an important direction for future work is to demonstrate how reference synthesis would be integrated in the implementation of behavior-preserving refactorings. We expect this to be relatively straightforward. Before applying the main refactoring transformation, references would be locked, effectively turning them into abstract references. The refactoring could then operate on this abstract representation, freely inserting, moving, and deleting code containing locked references without having to account for how their bindings will be preserved. Once the transformation is complete, reference synthesis would regenerate the concrete references, ensuring that the indented bindings are preserved.

4.10 Data Availability

The original publication is accompanied by an artifact (Pelsmaeker, Zwaan, et al., 2025b), a Docker container that includes our reference synthesis tool, source code, and its dependencies. Our tool can be executed in *evaluation* mode to reproduce the data presented in Section 4.6. The evaluation utilizes a set of test programs with locked references, which are all included in the artifact. Additionally, the original test sets from which the test cases were derived are also provided for further reference.

5

Conclusion

When developing a DSL, the bulk of the effort is typically spent on implementing its compiler, transpiler, or interpreter. However, practical adoption of a DSL also depends heavily on the availability of high-quality editor services, such as code completion, to help users read and write code more efficiently. Implementing editor services takes additional effort and often requires the semantics of the language to be partially reimplemented to suit the particular editor service. This duplication of effort increases both development and maintenance costs. As a result, the overall effort required to build a usable DSL limits the ease with which new DSLs can be introduced to solve problems in a specific domain. To reduce redundancy and lower the barrier to develop a DSL, we want to derive editor services automatically from the formal definition of the language. In this dissertation, we explored the main research question, which was:

> How can we derive editor services from the static-semantic specification of a programming language?

In this chapter, we will place this dissertation in the context of related work in Section 5.1. We then discuss the results and contributions of this work in Section 5.2, and conclude by outlining directions for future work in Section 5.3.

5.1 Related Work

This dissertation sits at the intersection of language workbenches, constraint-based semantic tooling, incomplete programs, and binding-preserving program transformations. Across these areas, a recurring objective is to reduce the amount of language-specific effort required to provide sophisticated tooling for programming languages.

Language workbenches (Fowler, 2005) such as Rascal (Klint, van der Storm, and Vinju, 2009; Klint, van der Storm, and Vinju, 2010; Klint, van der Storm, and Vinju, 2019), JastAdd (Ekman and Hedin, 2007), Xtext (Eysholdt and Behrens, 2010), and Spoofax (Kats and Visser, 2010b; Visser, Wachsmuth, et al., 2014) already derive a range of editor functionality from language definitions (Erdweg, van der Storm, Völter, Boersma, et al., 2013). Much of this automatically derived functionality is syntax-directed, including services such as syntax highlighting, code folding, and automatic formatting, while Omar, Voysey, Hilton, Sunshine, et al. (2017) noted that formal semantic foundations of editor services have received less attention so far. More sophisticated semantic editor services, therefore, often require additional language-specific implementation effort, even when the needed information is already represented in the language implementation.

Rascal's TypePal (Rascal, 2021) provides a framework for name analysis, type checking, and type inference, but does not automatically derive code completion. Xtext derives basic code completion from the language's grammar, but validating the completion entries against the language's semantics has to be implemented separately (Bettini, 2011). Similarly, Spoofax can synthesize valid syntactic code completion suggestions from the language's grammar, but does not derive semantic code completion (de Souza Amorim, Erdweg, et al., 2016). Semantic editor services in JastAdd can be derived by reusing parts of the existing semantic analysis already present in RAG-based compilers (Söderberg and Hedin, 2011). These approaches show the value in reusing language implementations, but differ in the extent to which the editor service itself is derived from the language specification. Our work builds on these ideas by reusing the language's declarative specification of name resolution and typing as input to language-parametric semantic editor services.

Relatedly, constraint-based approaches have also been used to express the correctness requirements of transformed programs. In refactoring literature, name and type constraints have been used to describe the invariants that must be preserved, after which a constraint solver is used to find a transformed program that satisfies those constraints (Steimann, 2018; Tip et al., 2011). Robust projectional editing similarly represents well-formedness as a constraint problem, computing edits that preserve or restore well-formedness (Steimann, Frenkel, and Voelter, 2017). These approaches are closely related to this work, as they not only check the program but also guide its construction or modification. The main difference

is that our approach uses a separation of concerns: the solver can already verify whether the program is correct without knowing the actual concrete names, any only synthesizes concrete names later.

Another connection is the use of holes or placeholders to represent incomplete programs. The structure editor of Hazelnut (Omar, Voysey, Hilton, Aldrich, et al., 2017) uses holes to represent incomplete programs and as a barrier between an expected type and the provided type, and focuses on edit actions that preserve the typing properties. Subsequent work explores the execution and program synthesis of programs with holes (Omar, Voysey, Chugh, et al., 2019; Lubin et al., 2020). Proof assistants such as Agda (Coquand, Takeyama, and Synek, 2006) use typed holes as goals for interactive synthesis. Spoofax implements syntactic code completion by representing an incomplete syntax tree using placeholders (de Souza Amorim, Erdweg, et al., 2016). Our work similarly uses holes to represent incomplete programs, but then uses them to drive a constraint solver to synthesize results for semantic editor services.

Preserving the name bindings in a program is a fundamental correctness concern of any refactoring, as modifying code may otherwise cause a reference to resolve to a different declaration. Previous work addressed this by detecting accidental name capture and repairing it either by renaming identifiers (Erdweg, van der Storm, and Dai, 2014), or by explicitly requalifying references in Java (Schäfer, Ekman, and de Moor, 2008). This dissertation builds on previous work where references in Java programs were "locked" to their declaration before the transformation, to be synthesized to concrete references afterward (Ekman, Schäfer, and Verbaere, 2008; Schäfer, Ekman, and de Moor, 2008; Schafer et al., 2012; Schaefer and de Moor, 2010). Our work extends this to apply to any language for which the typing and name-binding rules are defined using Statix (van Antwerpen, Bach, et al., 2018b).

Taken together, these lines of research show increasing interest in reusing formal or executable descriptions of a language to provide language tooling. This dissertation contributes to this direction by exploring how language-parametric editor services can be derived from declarative static-semantic specifications.

5.2 Discussion

The objective of this work was to explore how to automatically derive editor services from the static-semantic specification of a programming language. We did this in the context of Spoofax, using declarative static-semantic specifications written in the Statix meta-DSL as input to language-parametric editor services.

Towards Language-Parametric Semantic Editor Services In Chapter 2, we explored the high-level idea of automatically deriving sound editor services from the static-semantic

specification of a programming language written in the Statix meta-DSL. Statix is a constraint-based language for specifying the static semantics of a programming language, and has already been used to specify the semantics of a number of programming languages. We categorized the approach for five editor services: *reference resolution*, *find usages*, *list subclasses* (all in Section 2.4), *code completion* (Section 2.5), and *extract definition* (Section 2.6). For each editor service, we have shown how they can be expressed in terms of the rules and constraints from the static-semantic specification of the language. While Statix includes a constraint solver, we concluded that the current implementation is not suitable for use in editor services. The solver has to be extended to support constraints with holes, where user-defined criteria specify if and how the solver would try to find solutions for these holes. As part of this, the solver should support trying different alternatives, backtracking, and the ability to return multiple results.

Language-Parametric Static-Semantic Code Completion Code Completion is an editor service that can have many manifestations. The most basic code completion suggests words and phrases that occur in the rest of the document. A more useful code completion for programming languages proposes syntax from the DSL that is permissible in the current context (de Souza Amorim, Erdweg, et al., 2016). But a high-quality code completion editor service can also understand the semantics of the programming language, and can suggest names of declarations that occur elsewhere in the program that are type-sound in the current context. In Chapter 3, we derived such a code completion editor service from a language's static-semantic specification written in Statix.

While the Statix constraint solver can be used as-is to implement a type checker from a Statix semantic specification, for our purposes, we had to extend the implementation of the Statix solver to allow solving for partial constraints and returning multiple results. We used this to implement a search strategy and perform a limited form of program generation, where each result represents a valid code completion proposal. Each code completion proposal represents a program fragment that includes holes that may be further refined. To avoid generating infinitely many code completion proposals of arbitrarily large sizes, we limited the generated proposals to those that are expected from a code completion editor service. This allowed us to find type-safe code completion proposals, including syntax, identifiers, and function calls.

Language-Parametric Reference Synthesis There is a whole class of challenges that arise when performing program transformations. Under the assumption that the input program is (type-)correct, refactorings must be implemented such that they preserve the program's behavior. While there are many aspects specific to individual refactorings that are tricky to get right, it turns out that binding preservation is an aspect common to many

refactorings, such as those from the catalog of refactorings from Fowler (1999). Binding preservation is the idea that any occurrence of a name as a reference in a program must statically resolve to the same declaration before and after the transformation. This is not trivial, as program transformations can introduce new names or move code such that the same name suddenly resolves to a different declaration. In Chapter 4, we provide a solution as an *inverse lookup function*: given a current resolution scope and a target declaration, this function synthesizes a (qualified) reference that, when resolved, points from the scope to the intended declaration. The function is parameterized by the static-semantic specification of the language, requiring very little effort from the language developer to produce sound-by-construction reference synthesis. Reference synthesis can be used as a building block for other program transformations, of which renaming would be the simplest example.

Conclusion As we have shown, it is definitely possible to use the specification of a programming language to derive semantic editor services. Given an implementation of a language-parametric editor service and a static-semantic specification of a programming language, it is possible to derive these language-parametric editor services with relatively little implementation effort on the part of the language developer. Of course, most of the effort is instead shifted to the development of these language-parametric editor services. And while such editor services are reusable across programming languages, such a language-parametric implementation effort might not make sense if an editor service is only rarely used or very specific to a particular programming language.

The implementation of editor services should ideally be *sound* with respect to the static semantics of the language; that is, they must not introduce errors that would cause the type checker to reject the program. Therefore, our idea can be extended more generally: a programming language's specification models a language's behavior, and as such, if the derivation is sound, any editor service derived from it is sound by construction. However, it is difficult to prove that the derivation of the editor service is sound.

The main takeaway for language developers is that having a formal specification of the syntax and semantics of a programming language is always useful. At the very least, it provides guidance to the language users and developers on how the language is supposed to work. But the value of the specification becomes even more apparent when it is used as an input to language-parametric tests, tools, and editor services, where it can improve the correctness and reduce the implementation effort across languages.

5.3 Future Work

This dissertation has shown that static-semantic specifications can serve as a basis for deriving sound editor services. At the same time, the case studies revealed limitations in how these specifications are represented and queried. They also leave open the question of what other parts of a language specification could serve as a basis for deriving editor services.

A Queryable Static-Semantic Model This dissertation relies on Statix semantic specifications and the scope graphs they construct. Scope graphs provide a flexible model of the name-binding structure of a program. However, in the current implementation of Statix, there is a tight coupling between the constraint-based typing rules and the scope graph. While many name-binding concerns can be modeled using only a scope graph, others, including qualified references, are partly encoded in the typing rules. As a result, the current scope graph model does not by itself completely capture the name-binding structure of the program, and resolving a reference requires alternating between querying the scope graph and solving constraints. For editor services that need to reason about references, such as code completion, code navigation, or binding-preserving refactorings, this can create a large search space that can negatively impact performance, completeness, and usability of the derived editor services. Therefore, an important direction for future work is to investigate a static-semantic model that is better suited to queries from editor services, and in which the scope graph itself more completely models the name-binding structure of the program.

Refactorings Based on Reference Synthesis The reference synthesis algorithm developed in Chapter 4 is intended as a building block for binding-preserving program transformations. Future work should investigate how effectively this building block can be integrated into implementations of common refactorings that move, introduce, or rewrite references. Such an evaluation would help determine how much language-specific refactoring logic can be eliminated by deriving binding preservation from the language's static-semantic specification.

Dynamic Semantics This dissertation has focused exclusively on static semantics. However, useful editor services may also depend on program execution properties such as control and data flow. A formal specification of a language's dynamic semantics could therefore serve as another basis for automatically deriving language-parametric editor services. Examples include run-time services such as debuggers, read–eval–print loops (REPLs), as well as analyses that detect unreachable code or identify values that remain constant. Future work could investigate which classes of such services can be derived from dynamic-semantic specifications, and which additional analyses or representations are required to make these derivations practical.

Abstract Semantic Models Finally, an important direction for future work is to demonstrate how the reference synthesis building block would be integrated in the implementation of behavior-preserving refactorings. We expect this to be relatively straightforward. Before applying the main refactoring transformation, references would be locked. This effectively turns the concrete references into abstract references, much like how a parser turns concrete syntax into abstract syntax. The refactoring could then operate on this abstract representation. After the transformation is complete, reference synthesis would regenerate the concrete references, analogous to how an unparser or formatter reconstructs concrete syntax from an abstract representation.

Appendices

A

Language-Parametric Static-Semantic Code Completion

A.1 Syntax Definition of Micro-C# in SDF3

```
module microcs

imports Common

context-free start-symbols Start

context-free sorts Start

context-free syntax
  Start.Program = <
    <{ClassDecl "\n"}*>
  >

context-free sorts
  ClassDecl MemberDecl ParamDecl FinalStmt

context-free syntax
  ClassDecl.Class = <
    class <ID> {
        <{MemberDecl "\n"}*>
    }
  >

  ParamDecl.Param = <<Type> <ID>>

  MemberDecl.Field = <public <Type> <ID>;>
  MemberDecl.Method = <
    public <Type> <ID>(<{ParamDecl ", "}*>) {
        <{Stmt "\n"}*>
        <FinalStmt>
    }
  >

  FinalStmt.Return     = <return <Exp>;>
  FinalStmt.Throw      = <throw <Exp>;>
  FinalStmt.NoReturn   = <>

context-free sorts Stmt
context-free syntax
  Stmt.Block = < {
      <{Stmt "\n"}*>
  }>

  Stmt.DeclareTypedVar     = <<Type> <ID>;>
  Stmt.DeclareTypedVarInit = <<Type> <ID> = <Exp>;>
  Stmt.DeclareVarInit      = <var <ID> = <Exp>;>

  Stmt.Assign = <<LValue> = <Exp>;>
  Stmt.Nop    = <;>
  Stmt.Exp    = <<Exp>;>
```

```
context-free sorts Exp
context-free syntax
  Exp.New    = <new <Type>(<>)>

  Exp.Null   = <null>
  Exp.True   = <true>
  Exp.False  = <false>
  Exp.This   = <this>

  Exp.Lt     = [[Exp] < [Exp]]  {left}
  Exp.Eq     = [[Exp] == [Exp]] {left}
  Exp.Neq    = [[Exp] != [Exp]] {left}
  Exp.And    = <<Exp> && <Exp>> {left}
  Exp.Not    = <!<Exp>>

  Exp.Int    = <<INT>>
  Exp.Mul    = <<Exp> * <Exp>> {left}
  Exp.Add    = <<Exp> + <Exp>> {left}

  Exp.Cond   = <<Exp> ? <Exp> : <Exp>>
  Exp        = <(<Exp>)> {bracket}
  Exp        = LValue

context-free sorts LValue
context-free syntax
  LValue.Member  = <<Exp>.<LExp>> {right}
  LValue         = LExp

context-free sorts LExp
context-free syntax
  LExp.Project   = <<MemAcc>.<LExp>> {right}
  LExp           = MemAcc

context-free sorts MemAcc
context-free syntax
  MemAcc.Var     = <<ID>>
  MemAcc.Call    = <<ID>(<{Exp ", "}*>)>

context-free priorities
  LValue.Member <0> .> Exp = LValue,
  LValue.Member <0> .> {
    Exp.New
    Exp.Eq
    Exp.And
    Exp.Not
    Exp.Mul
    Exp.Add
    Exp.Cond
  }
```

```
context-free sorts Type
context-free syntax
  Type.VoidType   = <void>
  Type.IntType    = <int>
  Type.BoolType   = <bool>
  Type.ObjectType = <object>
  Type.ClassType  = <<ID>>

context-free priorities
  {
    Exp.New
    LExp.Project
    MemAcc.Var
    MemAcc.Call
  } > {
    Exp.Not
  } > {
    Exp.Mul
  } > {
    Exp.Add
  } > {
    Exp.Lt
  } > { left:
    Exp.Eq
    Exp.Neq
  } > {
    Exp.And
  } > {
    Exp.Cond
  }

template options
  ID = keyword {reject}
  keyword -/- [A-Za-z0-9]
```

A.2 Static Semantics of Micro-C# in Statix

```
module statics

imports
  signatures/microcs-sig

signature
  sorts TYPE
  constructors
    CLASS       : string * scope -> TYPE
    METHOD      : list(TYPE) * TYPE -> TYPE
    INT         : TYPE
    BOOL        : TYPE
    OBJECT      : TYPE
    NULL        : TYPE
    VOID        : TYPE
    BOTTOM      : TYPE
  sorts
    DECL = (path * (occurrence * TYPE))

signature
  namespaces
    Class       : ID
    Member      : ID
    Var         : ID

  relations
    type        : occurrence -> TYPE

  name-resolution
    labels
      P     // lexical parent

rules programOk : Start
  programOk(Program(classes)) :- {s}
    new s,
    init(s),
    classesOk(s, classes).

rules init : scope
  init(s) :-
    classOk(s, Class("Exception", [])).

rules classOk : scope * ClassDecl
  classesOk maps classOk(*, list(*))

  classOk(s, Class(name, members)) :- {cs T}
    new cs, cs -P-> s,
    T == CLASS(name, cs),
    declareClass(s, name, T),
    declareVarOccurrence(cs, Var{"this"@name}, T),
    membersOk(cs, members).

rules memberOk : scope * MemberDecl
  membersOk maps memberOk(*, list(*))

  memberOk(cs, Field(type, name)) :- {T}
    typeOfType(cs, type) == T,
    declareMember(cs, name, T).

  memberOk(cs, Method(returnType, name, params,
      stmts, finalStmt)) :- {ms RT PTS RTE T}
    new ms, ms -P-> cs,
    typeOfType(cs, returnType) == RT,
    typesOfParams(ms, params) == PTS,
    declareMember(cs, name, METHOD(PTS, RT)),
    finalStmtOk(ms, finalStmt, RT),
    stmtsOk(ms, stmts).

rules typeOfParam : scope * ParamDecl -> TYPE
  typesOfParams maps typeOfParam(*, list(*))
    = list(*)

  typeOfParam(s, Param(type, name)) = T :-
    typeOfType(s, type) == T,
    declareVar(s, name, T).

rules finalStmtOk : scope * FinalStmt * TYPE
  finalStmtOk(s, Return(e), T) :- {TE}
    typeOfExp(s, e) == TE,
    subtypeOf(TE, T).
  finalStmtOk(s, Throw(e), _) :- {T}
    typeOfExp(s, e) == T,
    subtypeOf(T, CLASS("Exception", _)).
  finalStmtOk(s, NoReturn(), VOID()).

rules stmtOk : scope * Stmt
  stmtsOk maps stmtOk(*, list(*))

  stmtOk(bs, Block(stmts)) :- {nbs}
    new nbs, nbs -P-> bs,
    stmtsOk(nbs, stmts).

  stmtOk(_, Nop()).

  stmtOk(bs, Exp(e)) :-
    typeOfExp(bs, e) == _.

  stmtOk(bs, DeclareTypedVar(type, name)) :- {T}
    typeOfType(bs, type) == T,
    declareVar(bs, name, T).
```

```
  stmtOk(bs, DeclareTypedVarInit(type, name, e))
      :- {T TE}
    typeOfType(bs, type) == T,
    subtypeOf(typeOfExp(bs, e), TE),
    subtypeOf(TE, T),
    declareVar(bs, name, T).

  stmtOk(bs, DeclareVarInit(name, e)) :- {T}
    typeOfExp(bs, e) == T,
    declareVar(bs, name, T).

  stmtOk(bs, Assign(lvalue, e)) :- {T TE}
    typeOfLValue(bs, lvalue) == T,
    subtypeOf(typeOfExp(bs, e), TE),
    subtypeOf(TE, T).

rules typeOfExp : scope * Exp -> TYPE
  typesOfExps maps typeOfExp(*, list(*)) = list(*)

  typeOfExp(s, LValue2Exp(lvalue)) =
    typeOfLValue(s, lvalue).

  typeOfExp(s, e@New(_)) =
    typeOfNew(s, e).

  typeOfExp(s, e@New(ClassType(_))) =
    typeOfNew(s, e).

  typeOfExp(_, Null())  = NULL().
  typeOfExp(_, True())  = BOOL().
  typeOfExp(_, False()) = BOOL().

  typeOfExp(s, This()) = typeOfVar(s, "this").

  typeOfExp(s, Lt(e1, e2)) = BOOL() :-
    subtypeOf(typeOfExp(s, e1), INT()),
    subtypeOf(typeOfExp(s, e2), INT()).

  typeOfExp(s, Eq(e1, e2)) = BOOL() :-
    typeOfExp(s, e1) == _,
    typeOfExp(s, e2) == _.

  typeOfExp(s, Neq(e1, e2)) = BOOL() :-
    typeOfExp(s, e1) == _,
    typeOfExp(s, e2) == _.

  typeOfExp(s, And(e1, e2)) = BOOL() :-
    subtypeOf(typeOfExp(s, e1), BOOL()),
    subtypeOf(typeOfExp(s, e2), BOOL()).

  typeOfExp(s, Not(e1)) = BOOL() :-
    subtypeOf(typeOfExp(s, e1), BOOL()).

  typeOfExp(s, Int(_)) = INT().

  typeOfExp(s, Mul(e1, e2)) = INT() :-
    subtypeOf(typeOfExp(s, e1), INT()),
    subtypeOf(typeOfExp(s, e2), INT()).

  typeOfExp(s, Add(e1, e2)) = INT() :-
    subtypeOf(typeOfExp(s, e1), INT()),
    subtypeOf(typeOfExp(s, e2), INT()).

  typeOfExp(s, Cond(ec, e1, e2)) = T :- {T1 T2}
    subtypeOf(typeOfExp(s, ec), BOOL()),
    typeOfExp(s, e1) == T1,
    typeOfExp(s, e2) == T2,
    lub(T1, T2) == T.

rules typeOfNew : scope * Exp -> TYPE

  typeOfNew(s, New(type)) = T :- {cs name}
    typeOfType(s, type) == T@CLASS(name, cs).

rules typeOfLValue : scope * LValue -> TYPE

  typeOfLValue(s, m@Member(_, _)) =
    typeOfMemberExp(s, m).

  typeOfLValue(s, LExp2LValue(MemAcc2LExp(
      Var(name)))) = T :-
    typeOfVarOrMember(s, name) == T.

  typeOfLValue(s, LExp2LValue(le)) =
    typeOfLExp(s, s, le).

rules typeOfMemberExp : scope * LValue -> TYPE

  typeOfMemberExp(s, Member(e, le)) = T :- {cs}
    typeOfExp(s, e) == CLASS(_, cs),
    typeOfLExp(s, cs, le) == T.

rules typeOfLExp : scope * scope * LExp -> TYPE

  typeOfLExp(s, cs, MemAcc2LExp(ma)) =
    typeOfMemAcc(s, cs, ma).

  typeOfLExp(s, cs, lv@Project(_, _)) =
    typeOfProjectExp(s, cs, lv).

  typeOfLExp(s, cs, lv@Project(Var(_), _)) =
    typeOfProjectExp(s, cs, lv).

  typeOfLExp(s, cs, lv@Project(Call(_, _), _)) =
    typeOfProjectExp(s, cs, lv).
```

A

```
rules typeOfProjectExp : scope * scope * LExp
    -> TYPE

  typeOfProjectExp(s, cs, Project(ma, le)) = T :-
      {T1 cs2}
    typeOfMemAcc(s, cs, ma) == CLASS(_, cs2),
    typeOfLExp(s, cs2, le) == T.

rules typeOfMemAcc : scope * scope * MemAcc
  -> TYPE

  typeOfMemAcc(s, cs, Var(name)) = T :-
    typeOfMember(cs, name) == T.

  typeOfMemAcc(s, cs, Call(name, args)) = TR :-
      {TA}
    typeOfMember(cs, name) == METHOD(TA, TR),
    typesOfExps(s, args) == TA.

// Classes
rules declareClass : scope * ID * TYPE
  declareClass(s, decl, T) :-
    !type[Class{decl@decl}, T] in s.

rules resolveClass : scope * ID -> list(DECL)
  resolveClass(s, name) = ps :-
    query type
      filter P*
      and { d' :- d' == Class{name@_} }
      min $ < P
      in s |-> ps.

rules typeOfClass : scope * ID -> TYPE
  typeOfClass(s, name) = T :- {decl}
    resolveClass(s, name) == [(_, (decl, T))|_].

// Members
rules declareMember : scope * ID * TYPE
  declareMember(cs, decl, T) :-
    !type[Member{decl@decl}, T] in cs.

rules resolveMember : scope * ID
    -> list((path * (occurrence * TYPE)))
  resolveMember(s, name) = ps :-
    query type
      filter P*
      and { d' :- d' == Member{name@_} }
      min $ < P
      in s |-> ps.

rules typeOfMember : scope * ID -> TYPE
  typeOfMember(s, name) = T :- {decl}
    resolveMember(s, name) == [(_, (decl, T))|_].
```

```
// Variables
rules declareVarOccurrence : scope * occurrence
    * TYPE
  declareVarOccurrence(s, occ@Var{name@_}, T) :-
    !type[occ, T] in s.

rules declareVar : scope * ID * TYPE
  declareVar(s, decl, T) :-
    declareVarOccurrence(s, Var{decl@decl}, T).

rules resolveVar : scope * ID -> list(DECL)
  resolveVar(s, name) = ps :-
    query type
      filter P*
      and { d' :- d' == Var{name@_} }
      min $ < P
      in s |-> ps.

rules typeOfVar : scope * ID -> TYPE
  typeOfVar(s, name) = T :- {decl}
    resolveVar(s, name) == [(_, (decl, T))|_].

// Types
rules typeOfType : scope * Type -> TYPE
  typeOfType(s, t@IntType())    = INT().
  typeOfType(s, t@BoolType())   = BOOL().
  typeOfType(s, t@ObjectType()) = OBJECT().
  typeOfType(s, t@VoidType())   = VOID().

  typeOfType(s, t@ClassType(name)) = T :-
    typeOfClass(s, name) == T@CLASS(_, _).

rules lub : TYPE * TYPE -> TYPE
  lub(T@CLASS(_, _), NULL())    = T.
  lub(NULL(), T@CLASS(_, _))    = T.
  lub(T,      T)                = T.

rules subtypeOf : TYPE * TYPE
  subtypeOf(T, T).
  subtypeOf(NULL(), T@CLASS(_, _)).
  subtypeOf(BOTTOM(), _).

// Complex Resolution
rules typeOfVarOrMember : scope * ID -> TYPE
  typeOfVarOrMember(s, name) = T :- {decl}
    // resolution order:
    // 1. variables
    // 2. member
    resolveVarOrMember(s, name)
      == [(_, (decl, T))|_].
```

```
329 rules resolveVarOrMember : scope * ID
330     -> list(DECL)
331   resolveVarOrMember(s, name)
332     = unwrapVarOrResolveMember(s, name,
333       resolveVar(s, name)).
334
335 rules unwrapVarOrResolveMember : scope * ID
336     * list(DECL) -> list(DECL)
337
338   unwrapVarOrResolveMember(_, _, ps@[(_,
339     (Var{decl@_}, T))|_]) = ps.
340
341   unwrapVarOrResolveMember(s, name, []) =
342     resolveMember(s, name).
343
344 // Placeholders
345 rules
346   programOk(Start-Plhdr()).
347   classOk(_, ClassDecl-Plhdr()).
348   memberOk(_, MemberDecl-Plhdr()).
349   typeOfParam(_, ParamDecl-Plhdr()) = BOTTOM().
350   stmtOk(_, Stmt-Plhdr()).
351   typeOfExp(_, Exp-Plhdr()) = BOTTOM().
352   typeOfLValue(_, LValue-Plhdr()) = BOTTOM().
353   typeOfNew(_, Exp-Plhdr()) = BOTTOM().
354   typeOfType(_, Type-Plhdr()) = BOTTOM().
355
```

A.3 Syntax Definition of Tiger in SDF3

```
module Tiger

imports Common

context-free start-symbols Module

sorts Module
context-free syntax

  Module.Mod = Exp

sorts Var Exp LValue
context-free syntax

  Var.Var = ID

  LValue = Var
  LValue.Subscript = <<LValue>[<Exp>]>
  LValue.FieldVar = <<LValue>.<ID>>

sorts Dec
context-free syntax

  Dec.VarDec = <var <ID> : <Type> := <Exp>>
  Dec.VarDecNoType = <var <ID> := <Exp>>
  Dec.FunDecs = <<{FunDec "\n"}+>> {longest-match}

context-free syntax

  Exp = LValue

  Exp.Let = <
    let
      <{Dec "\n"}*>
     in
      <{Exp ";\n"}*>
    end
  >

  Exp.Seq = <
    (
      <{Exp ";\n"}*>
    )
  >

  Exp.If = <
    if <Exp> then
      <Exp>
    else
      <Exp>
  >

  Exp.IfThen = <
    if <Exp> then
      <Exp>
  >

  Exp.While = <
    while <Exp> do
      <Exp>
  >

  Exp.For = <
    for <Var> := <Exp> to <Exp> do
      <Exp>
  >

  Exp.Break = <break>

  Exp.Assign = <<LValue> := <Exp>>

  Exp.NilExp = <nil>

  Exp.Array = <<ID>[<Exp>] of <Exp>>

  Exp.Record = <<ID>{ <{InitField ", "}*> }>

  Exp.Call = <<ID>(<{Exp ", "}*>)>

  Exp.Int     = IntConst

  Exp.Uminus  = [- [Exp]]
  Exp.Times   = [[Exp] * [Exp]]   {left}
  Exp.Divide  = [[Exp] / [Exp]]   {left}
  Exp.Plus    = [[Exp] + [Exp]]   {left}
  Exp.Minus   = [[Exp] - [Exp]]   {left}

  Exp.Eq      = [[Exp] = [Exp]]   {non-assoc}
  Exp.Neq     = [[Exp] <> [Exp]]  {non-assoc}
  Exp.Gt      = [[Exp] > [Exp]]   {non-assoc}
  Exp.Lt      = [[Exp] < [Exp]]   {non-assoc}
  Exp.Geq     = [[Exp] ⩾ [Exp]]   {non-assoc}
  Exp.Leq     = [[Exp] ⩽ [Exp]]   {non-assoc}

  Exp.And     = [[Exp] & [Exp]]   {left}
  Exp.Or      = [[Exp] | [Exp]]   {left}

  Exp.String  = StrConst
```

```
sorts Declarations
context-free syntax

  Declarations.Declarations = <
    declarations <{Dec "\n"}*>
  >

sorts FunDec FArg
context-free syntax

  FunDec.ProcDec = <
    function <ID>(<{FArg ", "}*>) =
      <Exp>
  >

  FunDec.FunDec = <
    function <ID>(<{FArg ", "}*>) : <Type> =
      <Exp>
  >

  FArg.FArg = <<ID> : <Type>>

// Type declarations
sorts TypeDec
context-free syntax

  Dec.TypeDecs = <<{TypeDec "\n"}+>>
    {longest-match}

  TypeDec.TypeDec = <type <ID> = <Type>>

// Type expressions
sorts Type
context-free syntax

  Type.Tid = ID
  Type.ArrayTy = <array of <ID>>
  Type.RecordTy = <
    {
       <{Field ", \n"}*>
    }
  >

// Records
sorts Field InitField
context-free syntax
  Field.Field = <<ID> : <ID>>

  InitField.InitField = <<ID> = <Exp>>

lexical sorts IntConst
lexical syntax

  IntConst = [0-9]+


lexical restrictions

  // Ensure greedy matching for lexicals

  IntConst  -/- [0-9]

lexical sorts StrConst
lexical syntax

  StrConst = "\"" StrChar* "\""
  StrChar = ~[\\\"\r\n]
  StrChar = [\\] [a]
  StrChar = [\\] [b]
  StrChar = [\\] [f]
  StrChar = [\\] [n]
  StrChar = [\\] [r]
  StrChar = [\\] [t]
  StrChar = [\\] [v]
  StrChar = [\\] [\^] [A-Z]
  StrChar = [\\] [0-9] [0-9] [0-9]
  StrChar = [\\] [x] [0-9a-fA-F] [0-9a-fA-F]
  StrChar = [\\] [\\]
  StrChar = [\\] [\"]

  StrChar = [\\] [\ \t\r\n]+ [\\]

lexical syntax

  ID = [a-zA-Z] [a-zA-Z0-9\_]*
  ID = "_main"

lexical restrictions

  ID -/- [a-zA-Z0-9\_]

lexical syntax

  ID = "label"        {reject}
  ID = "goto"         {reject}
  ID = "declarations" {reject}
  ID = "true"         {reject}
  ID = "false"        {reject}

lexical syntax

  ID = "array"        {reject}
  ID = "if"           {reject}
  ID = "then"         {reject}
  ID = "else"         {reject}
  ID = "while"        {reject}
  ID = "for"          {reject}
  ID = "to"           {reject}
  ID = "do"           {reject}
  ID = "let"          {reject}
  ID = "in"           {reject}
  ID = "end"          {reject}
```

```
  ID = "of"          {reject}
  ID = "break"       {reject}
  ID = "nil"         {reject}
  ID = "function"    {reject}
  ID = "var"         {reject}
  ID = "type"        {reject}
  ID = "import"      {reject}
  ID = "primitive"   {reject}

lexical syntax

  ID = "class"       {reject}
  ID = "extends"     {reject}
  ID = "method"      {reject}
  ID = "new"         {reject}

context-free priorities

  {
    Exp.Assign
  } > { right:
    Exp.If
    Exp.IfThen
    Exp.While
    Exp.For
  },

  // Precedence of operators: Unary minus has the
  // highest precedence. The operators *, / have
  // the next highest (tightest binding)
  // precedence, followed by +, -, then by
  // =, <>, >, <, ≥, ≤, then by &, then by |.

  // Associativity of operators: The operators
  // *, /, +, - are all left associative. The
  // comparison operators do not associate, so
  // a = b = c is not a legal expression,
  // a = (b = c) is legal.
```

```
  {Exp.Uminus}
  > {left :
    Exp.Times
    Exp.Divide}
  > {left :
    Exp.Plus
    Exp.Minus}
  > {non-assoc :
    Exp.Eq
    Exp.Neq
    Exp.Gt
    Exp.Lt
    Exp.Geq
    Exp.Leq}
  > Exp.And
  > Exp.Or,

  {
    Exp.Or
  } > {
    Exp.Array
  } > {
    Exp.Assign
  },

  {
    Exp.Uminus
    LValue.FieldVar
    LValue.Subscript
  } > {left:
    Exp.Times
    Exp.Divide
  }
```

A.4 Static Semantics of Tiger in Statix

```
module statics

imports
  signatures/tiger-sig

rules // single-file entry point

  programOk : Module

  programOk(Mod(e)) :- {s T}
    new s, init(s),
    typeOfExp(s, e) == T.

rules // multi-file entry point

//  projectOk : scope
//  projectOk(s).
//  fileOk : scope * Start
//  fileOk(s, Empty()).

signature // names
  namespaces
    Loop  : string
    Field : string

signature // names
  name-resolution
    labels R
    resolve Loop  filter P* min $ < P
    resolve Field filter P* min $ < P


signature
  relations
    typeOfDecl : occurrence -> TYPE
  namespaces
    Var  : string
    Type : string
  name-resolution
    labels P
    resolve Var  filter P* min $ < P
    resolve Type filter P* min $ < P


rules // type binding

  declareType : scope * string * TYPE

  declareType(s, x, T) :-
    s -> Type{x} with typeOfDecl T,
    // declaration is distinct
    typeOfDecl of Type{x} in s |-> [(_, (_, T))],
    @x.decl := x.

  declareVar : scope * string * TYPE

  declareVar(s, x, T) :-
    s -> Var{x} with typeOfDecl T,
    // declaration is distinct
    typeOfDecl of Var{x} in s |-> [(_, (_, T))],
    @x.decl := x.



  declareTypeRaw : scope * string * TYPE
  lookupType : scope * string -> TYPE

  declareTypeRaw(s, x, T) :-
    s -> Type{x} with typeOfDecl T,
    // declaration is distinct
    typeOfDecl of Type{x} in s |-> [(_, (_, T))],
    @x.decl := x.

  lookupType(s, x) = T :- {x'} @x.ref := x',
    typeOfDecl of Type{x} in s |->
      [(_, (Type{x'}, T))].

rules // variable binding

  declareVarRaw : scope * string * TYPE
  lookupVar : scope * string -> TYPE

  declareVarRaw(s, x, T) :-
    s -> Var{x} with typeOfDecl T,
    // declaration is distinct
    typeOfDecl of Var{x} in s |-> [(_, (_, T))],
    @x.decl := x.

  lookupVar(s, x) = T :- {x'} @x.ref := x',
    typeOfDecl of Var{x} in s |->
      [(_, (Var{x'}, T))].
```

```
rules // field binding

  declareField : scope * ID * TYPE
  lookupField  : scope * string -> TYPE

  declareField(s, x, T) :-
    s -> Field{x} with typeOfDecl T,
    // declaration is distinct
    typeOfDecl of Field{x} in s |-> [(_, (_, T))],
    @x.decl := x.

  lookupField(s, x) = T :- {x'} @x.ref := x',
    typeOfDecl of Field{x} in s |->
      [(_, (Field{x'}, T))].

signature
  sorts TYPE
  constructors
    UNIT   : TYPE
    INT    : TYPE
    STRING : TYPE
    NIL    : TYPE
    RECORD : scope -> TYPE
    ARRAY  : TYPE * scope -> TYPE
    FUN    : list(TYPE) * TYPE -> TYPE

rules subtype : TYPE * TYPE

  subtype(T, T).

  subtype(NIL(), RECORD(s)).

rules subtype0 : Type * TYPE * TYPE

  subtype0(t, T, T).

  subtype0(t, NIL(), RECORD(s)).

rules subtypes : list(TYPE) * list(TYPE)

  subtypes([], []).
  subtypes([T | Ts], [S | Ss]) :-
    subtype(T, S),
    subtypes(Ts, Ss).

rules equitype : TYPE * TYPE

  equitype(T, T).
  equitype(NIL(), RECORD(s)).
  equitype(RECORD(s), NIL()).

  lub : TYPE * TYPE -> TYPE
  lub(S, T) = S.
  lub(NIL(), T) = T.
```

```
rules

  typeOfExp  : scope * Exp -> TYPE
  typeOfExps   maps typeOfExp(*, list(*))
    = list(*)


  typeOfSeq  : scope * list(Exp) -> TYPE
  typeOfLVal : scope * LValue -> TYPE

rules

  typeOfExp(s_outer, Let(ds, es)) = T :- {s_body}
    new s_body,
    decsOk(s_outer, s_body, ds),
    typeOfSeq(s_body, es) == T.


  decsOk     : scope * scope * list(Dec)
  decOk      : scope * scope * Dec

  decsOk(s_outer, s_body, []) :-
    s_body -P-> s_outer.

  decsOk(s_outer, s_body, [dec]) :-
    s_body -P-> s_outer,
    decOk(s_body, s_outer, dec).

  decsOk(s_outer, s_body, [dec | decs@[_|_]]) :-
      {s_dec}
    new s_dec,
    s_dec -P-> s_outer,
    decOk(s_dec, s_outer, dec),
    decsOk(s_dec, s_body, decs).

rules // type declarations

  // Types: In the expression [let ..typedecs..
  // in exps end] the scope of a type identifier
  // starts at the beginning of the consecutive
  // sequence of type declarations defining it and
  // lasts until the end. The includes the headers
  // and bodies of any functions with the scope.

  // Name spaces: There are two different name
  // spaces: one for types, and one for functions
  // and variables. A type [a] can be "in scope"
  // at the same time as a variable [a] or a
  // function [a], but variables and functions of
  // the same name cannot both be in scope
  // simultaneously (one will hide the other).
```

```
  decOk(s, s_outer, TypeDecs(tdecs)) :-
    tdecsOk(s, s, tdecs).

  tdecOk  : scope * scope * TypeDec
  tdecsOk   maps tdecOk(*, *, list(*))

  tdecOk(s, s_outer, TypeDec(x, t)) :- {T}
    typeOfType(s_outer, t) == T,
    declareType(s, x, T).

   // note: type declarations in a sequence are
   // mutually recursive

rules // types

  typeOfType : scope * Type -> TYPE
  typeOfType(s, Tid(x)) = lookupType(s, x).

rules // function declarations

  // Parameters: In [function id(..id1: id2..)
  // = exp] the scope of the parameter id1 lasts
  // throughout the function body exp

  decOk(s, s_outer, FunDecs(fdecs)) :-
    fdecsOk(s, s_outer, fdecs).

  fdecsOk      maps fdecOk(*, *, list(*))
  fdecOk     : scope * scope * FunDec

  fdecOk(s, s_outer, d@ProcDec(f, args, e)) :-
      {s_fun Ts}
    new s_fun, s_fun -P-> s,
    typesOfArgs(s_fun, s_outer, args) == Ts,
    declareVar(s, f, FUN(Ts, UNIT())),
    typeOfExp(s_fun, e) == UNIT().

  fdecOk(s, s_outer, d@FunDec(f, args, t, e)) :-
      {s_fun Ts T S}
    new s_fun, s_fun -P-> s,
    typesOfArgs(s_fun, s_outer, args) == Ts,
    typeOfType(s_outer, t) == T,
    declareVar(s, f, FUN(Ts, T)),
    typeOfExp(s_fun, e) == S,
    subtypeO(t, S, T).

rules
  typeOfArg  : scope * scope * FArg -> TYPE
  typesOfArgs  maps typeOfArg(*, *, list(*))
    = list(*)

  typeOfArg(s_fun, s_outer, FArg(x, t)) = T :-
    typeOfType(s_outer, t) == T,
    declareVar(s_fun, x, T).
```

```
rules // function calls

  typeOfExp(s, Call(f, es)) = T :- {Ts Ss}
    lookupVar(s, f) == FUN(Ts, T),
    typeOfExps(s, es) == Ss,
    subtypes(Ss, Ts).

rules // variable declarations

  // Local variables: In the expression [let
  // ..vardec.. in exp end], the scope of the
  // declared variable starts just after its
  // vardec and lasts until the end.

  decOk(s, s_outer, VarDec(x, t, e)) :- {T S}
    typeOfType(s_outer, t) == T,
    typeOfExp(s_outer, e) == S,
    subtype(S, T),
    declareVar(s, x, T).

  decOk(s, s_outer, VarDecNoType(x, e)) :- {T}
    typeOfExp(s_outer, e) == T,
    T != NIL(),
    declareVar(s, x, T).

rules // arrays

  typeOfType(s, ArrayTy(x)) = ARRAY(T, s_arr) :-
    // unique token to distinguish the array type
    new s_arr,
    typeOfType(s, Tid(x)) == T.

rules // array creation

  typeOfExp(s, Array(x, e1, e2)) = ARRAY(T, s_arr)
      :- {S}
    typeOfType(s, Tid(x)) == ARRAY(S, s_arr),
    typeOfExp(s, e1) == INT(),
    typeOfExp(s, e2) == T,
    subtype(T, S).

rules // array indexing

  typeOfLVal(s, Subscript(e1, e2)) = T :- {s_arr}
    typeOfLVal(s, e1) == ARRAY(T, s_arr),
    typeOfExp(s, e2) == INT().

rules // statements

  typeOfExp(s, Assign(e1, e2)) = UNIT() :- {T S}
    typeOfLVal(s, e1) == T,
    typeOfExp(s, e2) == S,
    subtype(S, T).
```

```
rules

  typeOfLVal(s, Var2LValue(Var(x))) = T :-
    lookupVar(s, x) == T.

  typeOfExp(s, LValue2Exp(lval)) =
    typeOfLVal(s, lval).

  typeOfExp(s, LValue2Exp(Var2LValue(Var(x)))) =
    lookupVar(s, x).

rules // sequence

  typeOfSeq(s, []) = UNIT().

  typeOfSeq(s, [e]) = T :-
    typeOfExp(s, e) == T.

  typeOfSeq(s, [e | es@[_|_]]) = T :- {S}
    typeOfExp(s, e) == S,
    typeOfSeq(s, es) == T.

  typeOfExp(s, Seq(es)) = typeOfSeq(s, es).

  typeOfExp(s, If(e1, e2, e3)) = lub(T, S) :-
    typeOfExp(s, e1) == INT(),
    typeOfExp(s, e2) == T,
    typeOfExp(s, e3) == S,
    equitype(S, T).

  typeOfExp(s, IfThen(e1, e2)) = UNIT() :-
    typeOfExp(s, e1) == INT(),
    typeOfExp(s, e2) == UNIT().

  typeOfExp(s, While(e1, e2)) = UNIT() :- {s_loop}
    new s_loop, s_loop -P-> s,
    s_loop -> Loop{"break"} with
      typeOfDecl UNIT(),
    typeOfExp(s_loop, e1) == INT(),
    typeOfExp(s_loop, e2) == UNIT().

  typeOfExp(s, For(Var(x), e1, e2, e3)) = UNIT()
      :- {s_for}
    new s_for,
    s_for -P-> s,
    declareVar(s_for, x, INT()),
    s_for -> Loop{"break"} with typeOfDecl UNIT(),
    typeOfExp(s, e1) == INT(),
    typeOfExp(s, e2) == INT(),
    typeOfExp(s_for, e3) == UNIT().

  typeOfExp(s, Break()) = UNIT() :-
    typeOfDecl of Loop{"break"} in s
      |-> [(_, (_, UNIT()))].
```

```
rules // literals

  typeOfExp(s, Int(i)) = INT() :-
    @i.lit := i.

rules // operators

  typeOfExp(s, Uminus(e)) = INT() :-
    typeOfExp(s, e) == INT().

  typeOfExp(s, Divide(e1, e2)) = INT() :-
    typeOfExp(s, e1) == INT(),
    typeOfExp(s, e2) == INT().

  typeOfExp(s, Times(e1, e2)) = INT() :-
    typeOfExp(s, e1) == INT(),
    typeOfExp(s, e2) == INT().

  typeOfExp(s, Minus(e1, e2)) = INT() :-
    typeOfExp(s, e1) == INT(),
    typeOfExp(s, e2) == INT().

  typeOfExp(s, Plus(e1, e2)) = INT() :-
    typeOfExp(s, e1) == INT(),
    typeOfExp(s, e2) == INT().

  typeOfExp(s, Eq(e1, e2)) = INT() :- {T S}
    typeOfExp(s, e1) == T,
    typeOfExp(s, e2) == S,
    equitype(T, S).

  typeOfExp(s, Neq(e1, e2)) = INT() :- {T S}
    typeOfExp(s, e1) == T,
    typeOfExp(s, e2) == S,
    equitype(T, S).

  typeOfExp(s, Gt(e1, e2)) = INT() :-
    typeOfExp(s, e1) == INT(),
    typeOfExp(s, e2) == INT().

  typeOfExp(s, Lt(e1, e2)) = INT() :-
    typeOfExp(s, e1) == INT(),
    typeOfExp(s, e2) == INT().

  typeOfExp(s, Geq(e1, e2)) = INT() :-
    typeOfExp(s, e1) == INT(),
    typeOfExp(s, e2) == INT().

  typeOfExp(s, Leq(e1, e2)) = INT() :-
    typeOfExp(s, e1) == INT(),
    typeOfExp(s, e2) == INT().
```

```
  typeOfExp(s, Or(e1, e2)) = INT() :-
    typeOfExp(s, e1) == INT(),
    typeOfExp(s, e2) == INT().

  typeOfExp(s, And(e1, e2)) = INT() :-
    typeOfExp(s, e1) == INT(),
    typeOfExp(s, e2) == INT().

rules // record type

  typeOfType(s, RecordTy(fields))
      = RECORD(s_rec) :-
    new s_rec,
    fieldsOk(s_rec, s, fields).

  fieldOk : scope * scope * Field
  fieldsOk maps fieldOk(*, *, list(*))

  fieldOk(s_rec, s_outer, Field(x, t)) :- {T}
    typeOfType(s_outer, Tid(t)) == T,
    declareField(s_rec, x, T).

rules // literals

  typeOfExp(s, NilExp()) = NIL().

rules // record creation

  typeOfExp(s, e@Record(t, inits)) = RECORD(s_rec)
      :- {s_init ds}
    typeOfType(s, Tid(t)) == RECORD(s_rec),
    new s_init,
    initsOk(s, s_rec, s_init, inits),
    typeOfDecl of Field{_} in s_rec |-> ds,
    fieldsSameLength(ds, inits),
    allFieldsInitialized(t, ds, s_init).

  fieldsSameLength : list((path * (occurrence
    * TYPE))) * list(InitField)

  fieldsSameLength([],[]).

  fieldsSameLength([_|xs], [_|ys]) :-
    fieldsSameLength(xs, ys).

  initOk : scope * scope * scope * InitField
  initsOk maps initOk(*, *, *, list(*))

  initOk(s, s_rec, s_init, InitField(x, e)) :-
      {S T}
    lookupField(s_rec, x) == T,
    typeOfExp(s, e) == S,
    declareField(s_init, x, S),
    subtype(S, T).

  fieldInitialized : ID * (path * (occurrence
    * TYPE)) * scope
  allFieldsInitialized maps fieldInitialized(*,
    list(*), *)

  fieldInitialized(t, (_, (Field{x}, _)), s) :-
    typeOfDecl of Field{x} in s |-> [_].
    // t is passed such that error is displayed
    // be on t; noting that init of x is missing

rules // record field access

  typeOfLVal(s, FieldVar(lval, x)) = T :- {s_rec}
    typeOfLVal(s, lval) == RECORD(s_rec),
    lookupField(s_rec, x) == T.

rules // literals

  typeOfExp(s, String(v)) = STRING() :-
    @v.lit := v.

rules

  init : scope

  init(s) :-
    declareTypeRaw(s, "int",    INT()),
    declareTypeRaw(s, "string", STRING()),

    declareVarRaw(s, "print",
      FUN([STRING()], UNIT())),
    declareVarRaw(s, "chr",
      FUN([INT()], STRING())),
    declareVarRaw(s, "ord",
      FUN([STRING()], INT())),
    declareVarRaw(s, "size",
      FUN([STRING()], INT())),
    declareVarRaw(s, "substring",
      FUN([STRING(), INT(), INT()], STRING())),
    declareVarRaw(s, "concat",
      FUN([STRING(), STRING()], STRING())),
    declareVarRaw(s, "not",
      FUN([INT()], INT())),
    declareVarRaw(s, "exit",
      FUN([INT()], UNIT())),
    declareVarRaw(s, "getchar",
      FUN([], STRING())),
    declareVarRaw(s, "flush",
      FUN([], UNIT())),
    declareVarRaw(s, "printi",
      FUN([INT()], UNIT())).
```

A

```
553 rules // placeholders
554
555   programOk(Module-Plhdr()).
556
557   initOk(_, _, _, InitField-Plhdr()).
558
559   typeOfLVal(_, LValue-Plhdr()) = T.
560
561   decOk(_, _, Dec-Plhdr()).
562
563   tdecOk(_, _, TypeDec-Plhdr()).
564
565   typeOfType(_, Type-Plhdr()) = T.
566
567   fdecOk(_, _, FunDec-Plhdr()).
568
569   typeOfArg(_, _, FArg-Plhdr()) = T.
570
571   fieldOk(_, _, Field-Plhdr()).
572
573   typeOfExp(_, Exp-Plhdr()) = T.
```

A.5 Definitions of Strategy Combinators

```
id     :: a → [a]  -- Identity
seq    :: (a → [b]) → (b → [c]) → a → [c] -- Sequential composition
or     :: (a → [b]) → (a → [b]) → a → [b] -- Disjunction
try    :: (a → [a]) → a → [a]  -- Apply strategy if possible
repeat :: (a → [a]) → a → [a]  -- Repeat strategy until failure
fixSet :: Eq a ⇒ (a → [a]) → a → [a]  -- Repeat strategy until nothing changes
limit  :: Int → (a → [b]) → a → [b]  -- Limit sequence resulting from strategy
(.>)    = seq

fail   :: a → [b]  -- Failure
glc    :: (a → [b]) → (b → [c]) → (a → [c]) → a → [c]  -- Guarded left choice
single :: (a → [b]) → a → [b]  -- Assert that strategy returns a singleton

id i = [i]

fail _ = []

glc s1 s2 s3 a = case s1 a of
  [] → s3 a
  rs → concatMap s2 rs

seq s1 s2 = glc s1 s2 fail

or s1 s2 a = (s1 a) ++ (s2 a)

try s = glc s id id

repeat s a = try (seq s (repeat s)) a

fixSet s a = case s a of
  [] → []
  [b] → if a == b then [a] else fixSet s b
  rs → concatMap (fixSet s) rs

limit i s a = take i (s a)

single s a = case s a of
  [b] → [b]
  _ → []
```

B

Language-Parametric Reference Synthesis

B.1 Soundness Proofs

After some preliminaries (Appendix B.1.1), in this appendix, we prove theorem 4.3: that the reference synthesis algorithm only computes well-typed solutions (Appendix B.1.2). Next, we prove theorem 4.4: that the solutions it computes correspond to a composite path in the scope graph (Appendix B.1.3).

B.1.1 Preliminaries

Notation To simplify dealing with variable renaming and capture, for these proofs we assume Barendregt's convention. Where relevant, we present the constraint set in a solver state in the order we intend to solve them. Moreover, we are overloading some notation for brevity:

- We write $\kappa \rightarrowtail \kappa'$ to denote $\exists \theta\, H'.\, \kappa \rightarrowtail \theta, H' \wedge \kappa' = \langle\, \mathcal{G} \,|\, \overline{C} \,|\, U \,|\, H' \,\rangle\theta$
  (i.e., κ can be expanded into κ', ignoring other expansions).
- Similarly, we write $\kappa \twoheadrightarrow \kappa'$ to denote $\exists \overline{\kappa}'.\, \kappa \twoheadrightarrow \overline{\kappa}' \wedge \kappa' \in \overline{\kappa}'$
  (i.e., κ can step into κ', ignoring other steps).

We use the $\oplus$ operator to manipulate the components of a solver state:

$$\kappa \oplus C = \langle\, \mathcal{G} \,|\, \overline{C}; C \,|\, U \,|\, H \,\rangle$$

Finally, we use $\mathcal{G}_\kappa$, $\overline{C}_\kappa$, U_κ and H_κ to denote the components of a solver state κ, or $\mathcal{G}_i$, $\overline{C}_i$, U_i and H_i to denote the components of a solver state κ_i, and similar for primes.

In these proofs, we will often use exhaustive application of the step relations, written with a bullet symbol ($\bullet$) and defined as follows:

Definition B.1.1 (Exhaustive Application)**.**

$$\frac{\kappa_1 \to^* \kappa_2 \qquad \kappa \not\to}{\kappa_1 \to^\bullet \kappa_2}$$

and likewise for $\rightarrowtail^\bullet$ and $\twoheadrightarrow^\bullet$.

The $\kappa \not\to$ premise states that there are no further steps possible from κ, ensuring exhaustiveness. The $\to^\bullet$ relation only differs from the reflexive transitive closure regarding this additional premise. As such, we will freely convert between these relations when applicable.

Next, we define which initial constraints we consider well-formed:

Definition B.1.2 (Well-formed Initial Constraint)**.**

$$\frac{\forall \mathscr{P}\, \kappa.\ \left(FV(\mathscr{P}) = \varnothing \wedge \left\langle \varnothing \,\middle|\, P_0(\mathscr{P}) \,\middle|\, \varnothing \,\middle|\, \varnothing \right\rangle \to^\bullet \kappa\right) \implies \overline{C}_\kappa = \varnothing \vee \overline{C}_\kappa = \{\mathsf{false}\}}{\$ \vdash P_0\ \textbf{init}}$$

This specification states that, for any *ground* input program $\mathscr{P}$, the solver will either give an accepting configuration ($C_\kappa = \varnothing$), or a rejecting configuration ($C_\kappa = \{\mathsf{false}\}$). That is, we exclude stuckness from being a possible result of solving the initial constraint *on a ground program*.

Second, along the lines of definition 4.1 by Rouvoet et al. (2020), we define the embedding of a substitution θ as the embedding of the conjunction of the equalities that it represents:

Definition B.1.3 (Embedding of Subtitution)**.**

$$[\![\theta]\!] = \left(\mathop{\Large *}_{x \in \mathsf{dom}(\theta)} x \stackrel{?}{=} x\theta \right)$$

We use this embedding to use constraints and substitution interchangeably:

Lemma B.1.4 (Equivalence of Embedding and Applying Substitutions)**.**

$$\kappa_1 \oplus [\![\theta]\!] \to^* \kappa_2 \wedge \overline{C}_1\theta = \overline{C}_2 \iff \kappa_1\theta \approx \kappa_2$$

Proof. By induction on $\rightarrow$, using rules Op-Conj and Op-Eq-True, using the fact that unification with a free variable (such as generated by the embedding) on one side cannot fail. □

Finally, we state lemmas about weakening and strengthening the solver trace. First, we state that removing equality constraints from a successful solver trace cannot yield a failing solver traces (it either gets stuck, or it is successful, in which case the equality constraints were redundant):

Lemma B.1.5 (Equality Weakening)**.**

$$\forall t_1\, t_2\, \kappa_1\, \kappa_2\, \kappa_2'.\ \left(\kappa_1 \oplus (t_1 \stackrel{?}{=} t_2) \rightarrow^\bullet \kappa_2 \wedge \overline{C}_2 = \varnothing\right) \implies \left(\kappa_1 \rightarrow^\bullet \kappa_2' \wedge \overline{C}_2' \neq \{\mathsf{false}\}\right)$$

Proof. The presence of equality constraints cannot block other constraints from being solved.[1] Thus, if a failure state can be reached from the initial state on the right-hand side, it must have been reachable from the initial state on the left-hand side as well, by not selecting $t_1 \stackrel{?}{=} t_2$ to be solved. Then, by confluence, any trace for the initial state should reach the failure state. This contradicts the assumption that the first trace was successful. □

Second, we use the lemma that strengthening a solver trace with redundant equalities (i.e., equalities that hold in the final state anyway) preserves satisfiability:

Lemma B.1.6 (Strengthening with Redundant Equalities)**.**

$$\forall \kappa_1\, \kappa_2\, \theta.\ \kappa_1 \rightarrow^\bullet \kappa_2 \wedge \kappa_2\theta = \kappa_2 \implies \kappa_1 \oplus [\![\theta]\!] \rightarrow^* \kappa_2$$

Remark. This lemma assumes that the operational semantics tracks which variables we have unified and substituted already during constraint solving. Without this assumption, θ may contain information that contradicts κ_1, but (due to substitution) is not represented in κ_2. For example, when we solve a constraint $f(x) \stackrel{?}{=} f(g())$, for some variable x and term constructors f and g, we substitute x for $g()$, and remember in all future states that x was substituted for $g()$. The θ in the premise of the lemma below is assumed to be *consistent* with the variables that we have substituted for already, where by consistent we mean the following. θ' is consistent with θ when $\forall x \in \mathrm{dom}(\theta).\ x \in \mathrm{dom}(\theta') \implies \mathrm{mgu}(\theta(x), \theta'(x)) \neq \bot$. The operational semantics does not currently track information about which variables have been unified explicitly, but can be adapted to do so.

[1] This can be proved by the fact that the only negative condition on the constraint set occurs in the guard premise of the Op-Query rule. However, equality constraints do not affect guard, hence all constraints in $\overline{C}$ can still be solved. On the other hand, the guard premise prevents the general version of this lemma from being true. This is expected, as e.g., strengthening a constraint set with edge assertions could invalidate earlier queries.

Proof. The new equalities do not prevent $\overline{C}_1$ from being solved.[1] Thus, the following trace exists:

$$\kappa_1 \oplus [\![\theta]\!] \to^* \kappa_2 \oplus [\![\theta]\!]$$

Then, because equalities can be solved unconditionally, we can solve those next.

$$\kappa_2 \oplus [\![\theta]\!] \to^* \kappa_2' = \langle\, \mathcal{G}_2' \mid \overline{C}_2\theta \mid U_2' \mid H_2' \,\rangle$$

Then, by Theorem B.1.4, $\kappa_2' = \kappa_2\theta$. By the assumption that $\kappa_2\theta = \kappa_2$, we have that $\kappa_2' = \kappa_2$. Therefore, a trace $\kappa_1 \oplus [\![\theta]\!] \to^* \kappa_2$ exists. □

B.1.2 Generated Proposals are Well-Typed

Now, we prove our first soundness property: the algorithm only computes well-typed solutions.

Theorem (Soundness 1: Well-Typed Solutions)**.**

$$\begin{aligned}\forall \mathbb{S}\, P_0\, \mathscr{P}\, \mathscr{P}'.\, \mathbb{S} \vdash P_0 \ \textbf{init}\\ \implies \mathsf{synthesize}(\mathscr{P}) = \mathscr{P}'\\ \implies \exists \mathscr{G}.\, \mathbb{S} \vdash \langle\, \varnothing \mid P_0(\mathscr{P}') \mid \varnothing \mid \varnothing \,\rangle \to^\bullet \langle\, \mathscr{G} \mid \varnothing \mid \varnothing \mid \varnothing \,\rangle\end{aligned}$$

This property states that applying the initial constraint for any synthesized program $\mathscr{P}'$ is satisfiable; i.e., can be reduced to a *successful* solver state.

Proof. We will prove this theorem by transforming the synthesis trace for $\mathscr{P}$ in a valid solver trace for $\mathscr{P}'$. Observe that each valid synthesis can be considered a trace with the following shape:

$$\kappa_1 = \langle\, \varnothing \mid P_0(\mathscr{P}) \mid U_1 \mid H_1 \,\rangle \to^\bullet \kappa_2 \rightarrowtail \kappa_2' \twoheadrightarrow^\bullet \kappa_n = \langle\, \mathcal{G}_n \mid \varnothing \mid U_n \mid H_n \,\rangle$$

where $\kappa_2' = \kappa_2\theta_2$, given that θ_2 is the substitution that was computed in the expand step $\kappa_2 \rightarrowtail \kappa_2'$. By Theorem B.1.4, we can create a new trace as follows:

$$\kappa_2 \oplus [\![\theta_2]\!] \to^\bullet \kappa_2' \twoheadrightarrow^\bullet \kappa_n$$

By repeating this on all expand steps, we can create the following sequence of traces:

$$\kappa_1 \rightarrow^\bullet \kappa_2$$
$$\kappa_2 \oplus [\![\theta_2]\!] \rightarrow^\bullet \kappa_3$$
$$\dots$$
$$\kappa_{n-1} \oplus [\![\theta_{n-1}]\!] \rightarrow^\bullet \kappa_n$$

These traces are all valid solver traces (i.e., each step is valid according to one of the Op-* rules).

Next, we use θ_{result} from synthesize to eliminate all intermediate equalities $\theta_2 \dots \theta_{n-1}$. Because θ_{result} is derived from U_n and H_n, θ_{result} is redundant in κ_n (i.e., $\kappa_n \theta_{\text{result}} = \kappa_n$). When prepending the embedding of this unifier to the last solver trace, by Theorem B.1.6, we construct the following trace:[2]

$$\kappa_{n-1} \oplus [\![\theta_{n-1}]\!] \oplus [\![\theta_{\text{result}}]\!] \rightarrow^\bullet \kappa_n$$

Using Theorem B.1.5 (removing $[\![\theta_{n-1}]\!]$), Statix's confluence, and the fact that all variables in θ_{result} are free in the original constraint, we infer that $\kappa_{n-1} \oplus [\![\theta_{\text{result}}]\!]$ either steps to κ_4, or gets stuck. By applying the same reasoning to the previous traces, we can conclude that $\kappa_1 \oplus [\![\theta_{\text{result}}]\!]$ either steps to κ_4, or gets stuck.

Using confluence and Theorem B.1.4, together with the fact that $[\![\theta_{\text{result}}]\!]$ can be solved first, we infer the following trace exists:

$$\kappa_1 \oplus [\![\theta_{\text{result}}]\!] \rightarrow^* \kappa_1' = \langle\, \varnothing \mid \overline{C}_1 \theta_{\text{result}} \mid U_1' \mid H_1' \,\rangle \rightarrow^\bullet \kappa_n'$$

where $\kappa_n' = \kappa_n$ or some stuck state. Note that $\overline{C}_1 \theta_{\text{result}} = P_0(\mathscr{P}')$, and hence the intermediate state κ_1' is exactly the initial state for solving $P_0(\mathscr{P}')$.

Finally, by our assumption that the initial constraint cannot get stuck on a ground program, κ_n' cannot be stuck. Hence $\kappa_n' = \kappa_n$, which is a successful state. For that reason, we conclude that $\mathscr{P}'$ is well-typed.

□

[2] Note that the variables in θ_{result} are free in the original constraint. For that reason, assuming the state tracks non-free variables (see the remark at Theorem B.1.6), we can freely move around this constraint.

B.1.3 Synthesized References Correspond to Composite Paths

Next, we prove our second soundness property: the synthesized references correspond to a composite path to the target. First, we restate our definition of composite path as a judgment $\kappa \vdash s^* \text{ CP}$, stating that s^* is a composite path in κ.

Definition B.1.7 (Composite Path)**.**

$$\text{CP-Single}\ \frac{}{\kappa \vdash s \text{ CP}} \qquad \text{CP-Comp}\ \frac{q = s_1 \xrightarrow[o]{r} \lambda x.\, E \text{ is solved in } \kappa \qquad p \in \mathsf{Ans}(\mathcal{G}_\kappa, q) \qquad \big(\mathsf{tgt}(p) = s_2 \vee s_2 \in \rho_{\mathcal{G}_\kappa}(\mathsf{tgt}(p))\big) \qquad \kappa \vdash s_2 \cdot s^* \text{ CP}}{\kappa \vdash s_1 \cdot s_2 \cdot s^* \text{ CP}}$$

Remark. The phrase 'q is solved in κ' can be made precise by making the state transitions ($\rightarrow$) track solved constraints explicitly.

We use this definition to define well-formed hole states H:

Definition B.1.8 (Well-formed H)**.**

$$\text{H-WF}\ \frac{\forall (s^*, t) \in \mathsf{ran}(H_\kappa).\ \kappa \vdash s^* \text{ CP}}{\kappa\ \mathsf{wf}_H}$$

This definition states that a hole state H_κ is well-formed if for each hole state $(s^*, t) \in \mathsf{ran}(H_\kappa)$, s^* is a composite path in κ. With this definition of composite paths, we can state our second soundness property as follows.

Theorem B.1.9 (Soundness 2: Solutions correspond to Composite Paths)**.**

$$\forall \kappa_0\, \kappa_1.\ (\kappa_0 \textit{ initialized by } \mathsf{synthesize} \wedge \kappa_0 \twoheadrightarrow^\bullet \kappa_1) \implies \kappa_1\ \mathsf{wf}_H$$

This theorem states that every synthesis result yields hole states $(s^*, t) \in \mathsf{ran}(H_1)$ such that s^* forms a composite path. To prove this theorem, we need the lemma that $\rightarrow$ preserves hole state well-formedness.

Lemma B.1.10.

$$\forall \kappa_0\, \kappa_1.\ \kappa_0\ \mathsf{wf}_H \wedge \kappa_0 \rightarrow \kappa_1 \implies \kappa_1\ \mathsf{wf}_H$$

Proof. This can be proven by case analysis on $\rightarrow$, where none of the rules modify the hole state (apart from substitution, which does not affect the s^* component, as it is ground). □

Ideally, we would prove a similar lemma for $\twoheadrightarrow$ as well. However, this cannot be done, as Op-Expand-Query prepends a scope to the composite path in $H(h)$, for which the corresponding query is not yet solved. Instead, we prove the following lemma, which states that, after an expand-step, subsequent solving ensures that H will *eventually* be well-formed.

Lemma B.1.11.

$$\forall \kappa_0\, \kappa_1\, \kappa_2.\ \kappa_0\ \mathsf{wf}_H \wedge \kappa_0 \rightarrowtail \kappa_1 \rightarrow^\bullet \kappa_2 \implies \kappa_2\ \mathsf{wf}_H$$

Proof. This lemma can be proven by case analysis on the $\kappa_0 \rightarrowtail \kappa_1$ step:

- Op-Expand-Pred: This rule does not affect H, hence $\kappa_1\ \mathsf{wf}_H$ and by Theorem B.1.10, $\kappa_2\ \mathsf{wf}_H$.
- Op-Expand-Query: This rule expands a query $q = s' \xrightarrow[o]{r}\!\!\!\!\!\rightarrow \lambda x.\, E$. As a result, it prepends s' to the composite path $(s'' \cdot s^*, t) \in \mathsf{ran}(H_0)$. When proving $\kappa_1\ \mathsf{wf}_H$, by inversion using H-WF and CP-Comp, we have the following proof obligations:
  - q is solved in κ_1,
  - $p \in \mathsf{Ans}(\mathscr{G}_1, q) \wedge (\mathsf{tgt}(p) = s'' \vee s'' \in \rho_{\mathscr{G}_1}(\mathsf{tgt}(p)))$, and
  - $\kappa_1 \vdash s'' \cdot s^*$ cp.

  The second obligation is a direct consequence of the Op-Expand-Query rule, and the last obligation is a direct consequence of the $\kappa_0\ \mathsf{wf}_H$ assumption. However, the first premise is not yet satisfied, as Op-Expand-Query does not actually solve q.

  To prove that $\kappa_2\ \mathsf{wf}_H$, we need to show that q is solved in κ_2. To this end, observe that the premises of Op-Expand-Query entail the premises of Op-Query for q. Hence, q is not *blocked*; it *can* be solved in κ_1. Thus, a state κ_1' exists such that $\kappa_1 \rightarrow \kappa_1'$ and q is solved in κ_1'. For that reason, $\kappa_1'\ \mathsf{wf}_H$. Now, consider the state κ_2' obtained after solving κ_1' completely (i.e., $\kappa_1' \rightarrow^\bullet \kappa_2'$). By Theorem B.1.10, $\kappa_2'\ \mathsf{wf}_H$. Finally, by confluence of the operational semantics, $\kappa_2' \approx \kappa_2$, and hence $\kappa_2\ \mathsf{wf}_H$.

□

With these lemmas in place, we can prove theorem 4.4:

Proof. Observe that any synthesis trace $\kappa_0 \twoheadrightarrow^\bullet \kappa_n$ can be decomposed into parts that start with an expand step, and a sequence of solve steps:

$$\left[\kappa_{i-1} \rightarrowtail \kappa_i' \rightarrow^\bullet \kappa_i\right]_{i \in 1 \ldots n}$$

We can prove the theorem using induction over the segments $1 \ldots n$ of the synthesis trace:

- *Base case*: κ_0 wf_H. This case is trivial, as each value in H_0 is initialized to the target scope s_d, which is a composite path by CP-Single.
- *Inductive case*: Follows immediately from Theorem B.1.11.

This proves κ_n wf_H, and hence each synthesized reference corresponds to a composite path. □

Bibliography

Aho, Alfred V. and Jeffrey D. Ullman (1972). *The theory of parsing, translation, and compiling*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc. ISBN: 0-13-914556-7.

Aigner, Sebastian, Roman Elizarov, Svetlana Isakova, and Dmitry Jemerov (2024). *Kotlin in Action, Second Edition*. Simon and Schuster. ISBN: 9781617299605.

van Antwerpen, Hendrik, Casper Bach, Arjen Rouvoet, and Eelco Visser (2018a). "Case Studies for Article: Scopes as Types." In: *Artifact Digital Object Group* 2.OOPSLA. DOI: 10.1145/3276915.

van Antwerpen, Hendrik, Casper Bach, Arjen Rouvoet, and Eelco Visser (2018b). "Scopes as types." In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA, pp. 1–30. DOI: 10.1145/3276484.

van Antwerpen, Hendrik, Pierre Néron, Andrew Tolmach, Eelco Visser, and Guido Wachsmuth (2016). "A constraint language for static semantic analysis based on scope graphs." In: *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. Ed. by Martin Erwig and Tiark Rompf. ACM, pp. 49–60. DOI: 10.1145/2847538.2847543.

van Antwerpen, Hendrik and Eelco Visser (2021a). "Scope States (Artifact)." In: *DARTS* 7.2. DOI: 10.4230/darts.7.2.1.

van Antwerpen, Hendrik and Eelco Visser (2021b). "Scope States: Guarding Safety of Name Resolution in Parallel Type Checkers." In: *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*. Ed. by Anders Møller and Manu Sridharan. Vol. 194. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. DOI: 10.4230/lipics.ecoop.2021.1.

Appel, Andrew W. (2002). *Modern Compiler Implementation in Java, 2nd edition*. Cambridge University Press. ISBN: 0-521-82060-X.

Asaduzzaman, Muhammad, Chanchal K. Roy, Kevin A. Schneider, and Daqing Hou (2014). "Context-Sensitive Code Completion Tool for Better API Usability." In: *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*. IEEE, pp. 621–624. DOI: 10.1109/icsme.2014.110.

Asaduzzaman, Muhammad, Chanchal K. Roy, Kevin A. Schneider, and Daqing Hou (2016). "A Simple, Efficient, Context-sensitive Approach for Code Completion." In: *Journal of Software: Evolution and Process* 28.7, pp. 512–541. DOI: 10.1002/smr.1791.

Bach, Casper, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser (2018). "Intrinsically-typed definitional interpreters for imperative languages." In: *Proceedings of the ACM on Programming Languages* 2.POPL, pp. 1–34. DOI: 10.1145/3158104.

Backus, J. W., F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, M. Woodger, and Peter Naur (1960). "Report on the algorithmic language ALGOL 60." In: *Communications of the ACM* 3.5, pp. 299–311. DOI: 10.1145/367236.367262.

Bal, Henri E. and Dick Grune (1994). *Programming language essentials*. 1. print. Wokingham [u.a.]: Addison-Wesley. ISBN: 0-201-63179-2.

Barendregt, Hendrik Pieter (1984). *The Lambda Calculus - Its Syntax and Semantics*. North-Holland. ISBN: 978-0-444-86748-3.

Bayer, Michael (2012). "SQLAlchemy." In: *The architecture of open source applications* 2. URL: https://aosabook.org/en/v2/sqlalchemy.html.

Bettini, Lorenzo (2011). "A DSL for writing type systems for Xtext languages." In: *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*. PPPJ '11. ACM, pp. 31–40. DOI: 10.1145/2093157.2093163.

Brady, Edwin (2013). "Idris, a general-purpose dependently typed programming language: Design and implementation." In: *Journal of Functional Programming* 23.5, pp. 552–593. DOI: 10.1017/s095679681300018x.

Brooks, Frederick P. (1996). "Keynote address: Language design as design." In: *History of programming languages—II*, pp. 4–16. DOI: 10.1145/234286.1057806.

Brooks Jr., Frederick P. (1995). *The Mythical Man-Month (Anniversary ed.)* USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 9780201835953.

Bruch, Marcel, Martin Monperrus, and Mira Mezini (2009). "Learning from examples to improve code completion systems." In: *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*. Ed. by Hans van Vliet and ValÃ©rie Issarny. ACM, pp. 213–222. DOI: 10.1145/1595696.1595728.

Coquand, Catarina, Makoto Takeyama, and Dan Synek (2006). "An Emacs-Interface for Type-Directed Support for Constructing Proofs and Programs." In: *European Joint Conferences on Theory and Practice of Software, ENTCS*. Vol. 2.

Corbett, Robert Paul (1985). "Static Semantics and Compiler Error Recovery." PhD thesis. EECS Department, University of California, Berkeley. URL: https://www2.eecs.berkeley.edu/Pubs/TechRpts/1985/5514.html.

Denkers, Jasper, Marvin Brunner, Louis van Gool, and Eelco Visser (2021). "Configuration Space Exploration for Digital Printing Systems." In: *Software Engineering and Formal Methods - 19th International Conference, SEFM 2021, Virtual Event, December 6-10, 2021, Proceedings*. Ed. by Radu Calinescu and Corina S. Pasareanu. Vol. 13085. Lecture Notes in Computer Science. Springer International Publishing, pp. 423–442. DOI: 10.1007/978-3-030-92124-8_24.

van Deursen, Arie, Paul Klint, and Joost Visser (2000). "Domain-Specific Languages: An Annotated Bibliography." In: *ACM SIGPLAN Notices* 35.6, pp. 26–36. DOI: 10.1145/352029.352035.

Ekman, Torbjörn and Görel Hedin (2007). "The JastAdd extensible Java compiler." In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. Ed. by Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr. ACM, pp. 1–18. DOI: 10.1145/1297027.1297029.

Ekman, Torbjörn, Max Schäfer, and Mathieu Verbaere (2008). "Refactoring is not (yet) about transformation." In: *Second ACM Workshop on Refactoring Tools, WRT 2008, in conjunction with OOPSLA 2008, Nashville, TN, USA, October 19, 2008*. ACM, pp. 1–4. DOI: 10.1145/1636642.1636647.

Engström, Emelie, Margaret-Anne Storey, Per Runeson, Martin Höst, and Maria Teresa Baldassarre (2020). "How software engineering research aligns with design science: a review." In: *Empirical Software Engineering* 25.4, pp. 2630–2660. DOI: 10.1007/s10664-020-09818-7.

Erdweg, Sebastian, Lennart C. L. Kats, Tillmann Rendel, Christian Kästner, Klaus Ostermann, and Eelco Visser (2011). "Growing a language environment with editor libraries." In: *Generative Programming And Component Engineering, Proceedings of the 10th International Conference on Generative Programming and Component Engineering, GPCE 2011, Portland, Oregon, USA, October 22-24, 2011*. Ed. by Ewen Denney and Ulrik Pagh Schultz. ACM, pp. 167–176. DOI: 10.1145/2047862.2047891.

Erdweg, Sebastian, Tillmann Rendel, Christian Kästner, and Klaus Ostermann (2011). "SugarJ: library-based syntactic language extensibility." In: *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*. Ed. by Cristina Videira Lopes and Kathleen Fisher. ACM, pp. 391–406. DOI: 10.1145/2048066.2048099.

Erdweg, Sebastian, Tijs van der Storm, and Yi Dai (2014). "Capture-Avoiding and Hygienic Program Transformations." In: *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*. Ed. by Richard Jones. Vol. 8586. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 489–514. DOI: 10.1007/978-3-662-44202-9_20.

Erdweg, Sebastian, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido H. Wachsmuth, and Jimi van der Woning (2013). "The State of the Art in Language Workbenches - Conclusions from the Language Workbench Challenge." In: *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings*. Ed. by Martin Erwig, Richard F. Paige, and Eric Van Wyk. Vol. 8225. Lecture Notes in Computer Science. Springer International Publishing, pp. 197–217. DOI: 10.1007/978-3-319-02654-1_11.

Erdweg, Sebastian, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning (2015). "Evaluating and comparing language workbenches: Existing results and benchmarks for the future." In: *Computer Languages, Systems & Structures* 44, pp. 24–47. DOI: 10.1016/j.cl.2015.08.007.

Eysholdt, Moritz and Heiko Behrens (2010). "Xtext: implement your language faster than the quick and dirty way." In: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM, pp. 307–309. DOI: 10.1145/1869542.1869625.

Flatt, Matthew (2012). "Creating languages in Racket." In: *Communications of the ACM* 55.1, pp. 48–56. DOI: 10.1145/2063176.2063195.

Fowler, Martin (1999). *Refactoring - Improving the Design of Existing Code*. Addison-Wesley. ISBN: 978-0-201-48567-7.

Fowler, Martin (2005). *Language Workbenches: The Killer-App for Domain Specific Languages?* URL: https://www.martinfowler.com/articles/languageWorkbench.html (visited on 10/01/2026).

Fowler, Martin (2011). *Domain-Specific Languages*. Ed. by Rebecca Parsons. Upper Saddle River, NJ: Addison-Wesley. ISBN: 978-0-321-71294-3.

Gosling, James, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley (Feb. 2015). *The Java Language Specification - Java SE 8 Edition*. URL: https://docs.oracle.com/javase/specs/jls/se8/html/ (visited on 08/24/2023).

Griswold, William G. (1992). "Program Restructuring As an Aid to Software Maintenance." PhD thesis. Seattle, WA, USA: University of Washington. URL: https://cseweb.ucsd.edu/~wgg/Abstracts/griswold-thesis.html.

Groenewegen, Danny M., Elmer van Chastelet, Max M. de Krieger, and Daniel A. A. Pelsmaeker (2023). "Eating Your Own Dog Food: WebDSL Case Studies to Improve Academic Workflows." In: *Eelco Visser Commemorative Symposium (EVCS 2023)*. Ed. by Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann. Vol. 109. Open Access Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 13:1–13:11. DOI: 10.4230/oasics.evcs.2023.13.

Groenewegen, Danny M., Elmer van Chastelet, Max M. de Krieger, Daniel A. A. Pelsmaeker, and Craig Anslow (2023). "Conf Researchr: A Domain-Specific Content Management System for Managing Large Conference Websites." In: *Eelco Visser Commemorative Symposium (EVCS 2023)*. Ed. by Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann. Vol. 109. Open Access Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 12:1–12:6. DOI: 10.4230/oasics.evcs.2023.12.

Gulwani, Sumit, Oleksandr Polozov, and Rishabh Singh (2017). "Program Synthesis." In: *Foundations and Trends in Programming Languages* 4.1-2, pp. 1–119. DOI: 10.1561/2500000010.

Hellendoorn, Vincent J., Sebastian Proksch, Harald C. Gall, and Alberto Bacchelli (2019). "When code completion fails: a case study on real-world completions." In: *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. Ed. by Gunter Mussbacher, Joanne M. Atlee, and Tevfik Bultan. IEEE, pp. 960–970. DOI: 10.1109/icse.2019.00101.

Hevner, Alan R., Salvatore T. March, Jinsoo Park, and Sudha Ram (2004). "Design Science in Information Systems Research." In: *MIS Quarterly* 28.1, pp. 75–105.

IEEE (1987). "IEEE Standard VHDL Language Reference Manual." In: *IEEE Std 1076-1987*, pp. 1–218. DOI: 10.1109/ieeestd.1988.122645.

Igarashi, Atsushi, Benjamin C. Pierce, and Philip Wadler (2001). "Featherweight Java: a minimal core calculus for Java and GJ." In: *ACM Transactions on Programming Languages and Systems* 23.3, pp. 396–450. DOI: 10.1145/503502.503505.

JetBrains (2010). *YouTrack Issue PY-2104: Support Safe Delete refactoring for Python classes and functions*. URL: https://youtrack.jetbrains.com/issue/PY-2104/ (visited on 12/18/2025).

JetBrains (2018). *YouTrack Issue WEB-34698: Support moving static and instance class methods*. URL: https://youtrack.jetbrains.com/issue/WEB-34698/ (visited on 12/18/2025).

JetBrains (2025). *IntelliJ IDEA Supported Languages*. URL: https://www.jetbrains.com/help/idea/discover-intellij-idea.html#IntelliJ-IDEA-supported-languages (visited on 12/15/2025).

Jetbrains (2021). *Jetbrains MPS*. URL: https://jetbrains.com/mps/ (visited on 03/30/2021).

Johnson, Stephen C. (1975). "Yacc: Yet Another Compiler-Compiler." In: *Unix Programmer's Manual*.

de Jonge, Maartje and Eelco Visser (2012). "A language generic solution for name binding preservation in refactorings." In: *International Workshop on Language Descriptions, Tools, and Applications, LDTA '12, Tallinn, Estonia, March 31 - April 1, 2012*. Ed. by Anthony Sloane and Suzana Andova. ACM, pp. 1–8. DOI: 10.1145/2427048.2427050.

Juhár, Ján and Liberios Vokorokos (2015). "A review of source code projections in integrated development environments." In: *2015 Federated Conference on Computer Science and Information Systems, FedCSIS 2015, Lódz, Poland, September 13-16, 2015*. Ed. by Maria Ganzha, Leszek A. Maciaszek, and Marcin Paprzycki. Vol. 5. IEEE, pp. 923–927. DOI: 10.15439/2015f289.

Kats, Lennart C. L. and Eelco Visser (2010a). "The Spoofax language workbench." In: *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*. Ed. by William R. Cook, Siobhán Clarke, and Martin C. Rinard. ACM, pp. 237–238. DOI: 10.1145/1869542.1869592.

Kats, Lennart C. L. and Eelco Visser (2010b). "The Spoofax language workbench: rules for declarative specification of languages and IDEs." In: *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*. Ed. by William R. Cook, Siobhán Clarke, and Martin C. Rinard. Reno/Tahoe, Nevada: ACM, pp. 444–463. DOI: 10.1145/1869459.1869497.

Katzen, Sid (2001). "High-Level Language." In: *The Quintessential PIC Microcontroller*. London: Springer London, pp. 227–246. DOI: 10.1007/978-1-4471-3704-7_9.

Keidel, Sven, Wulf Pfeiffer, and Sebastian Erdweg (2016). "The IDE portability problem and its solution in Monto." In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31 - November 1, 2016*. Ed. by Tijs van der Storm, Emilie Balland, and Dániel Varró. ACM, pp. 152–162. DOI: 10.1145/2997364.2997368.

Klint, Paul, Tijs van der Storm, and Jurgen Vinju (2009). "RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation." In: *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, Edmonton, Alberta, Canada, September 20-21, 2009*. IEEE, pp. 168–177. DOI: 10.1109/scam.2009.28.

Klint, Paul, Tijs van der Storm, and Jurgen Vinju (2019). "Rascal, 10 Years Later." In: *19th International Working Conference on Source Code Analysis and Manipulation, SCAM 2019, Cleveland, OH, USA, September 30 - October 1, 2019*. IEEE, pp. 139–139. DOI: 10.1109/scam.2019.00023.

Klint, Paul, Tijs van der Storm, and Jurgen J. Vinju (2010). "EASY Meta-Programming with Rascal. Leveraging the Extract-Analyze-SYnthesize Paradigm for Meta-Programming." In: *Proceedings of the 3rd International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'09)*. LNCS. Springer. DOI: 10.5555/1949925.1949932.

Konat, Gabriël, Lennart Kats, Guido Wachsmuth, and Eelco Visser (2013). "Declarative Name Binding and Scope Rules." In: *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers*. Ed. by Krzysztof Czarnecki and Görel Hedin. Vol. 7745. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 311–331. DOI: 10.1007/978-3-642-36089-3_18.

Kosar, Tomaž, Marjan Mernik, and Jeff Carver (2011). "The impact of tools supported in integrated-development environments on program comprehension." In: *Proceedings of the International Conference on Information Technology Interfaces, ITI*.

Krahn, Holger, Bernhard Rumpe, and Steven Völkel (2010). "MontiCore: a framework for compositional development of domain specific languages." In: *International Journal on Software Tools for Technology Transfer* 12.5, pp. 353–372. DOI: 10.1007/s10009-010-0142-1.

Lamport, Leslie (1989). *LaTeX - A Document Preparation System*. Addison-Wesley. ISBN: 978-0-201-15790-1.

Li, Huiqing, Claus Reinke, and Simon Thompson (2003). "Tool support for refactoring functional programs." In: *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2003, Uppsala, Sweden, August 28, 2003*. ACM, pp. 27–38. DOI: 10.1145/871895.871899.

Li, Huiqing and Simon Thompson (2012). "Let's make refactoring tools user-extensible!" In: *Fifth Workshop on Refactoring Tools 2012, WRT '12, Rapperswil, Switzerland, June 1, 2012*. Ed. by Peter Sommerlad. ACM, pp. 32–39. DOI: 10.1145/2328876.2328881.

Li, Huiqing, Simon Thompson, László Lövei, Zoltán Horváth, Tamás Kozsik, Anikó Víg, and Tamás Nagy (2006). "Refactoring Erlang Programs." In: *The Proceedings of 12th International Erlang/OTP User Conference*. Stockholm, Sweden. URL: http://www.cs.kent.ac.uk/pubs/2006/2455.

Li, Huiqing, Simon Thompson, and Claus Reinke (2005). "The Haskell Refactorer, HaRe, and its API." In: *Electronic Notes in Theoretical Computer Science* 141.4, pp. 29–34. DOI: 10.1016/j.entcs.2005.02.053.

Lindblad, Fredrik and Marcin Benke (2006). "A Tool for Automated Theorem Proving in Agda." In: *Types for Proofs and Programs, International Workshop, TYPES 2004, Jouy-en-Josas, France, December 15-18, 2004, Revised Selected Papers*. Ed. by Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner. Vol. 3839. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 154–169. DOI: 10.1007/11617990_10.

van der Lippe, Tim, Thomas Smith, Daniel A. A. Pelsmaeker, and Eelco Visser (2016). "A scalable infrastructure for teaching concepts of programming languages in Scala with WebLab: an experience report." In: *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala*. SPLASH '16. ACM, pp. 65–74. DOI: 10.1145/2998392.2998402.

Lubin, Justin, Nick Collins, Cyrus Omar, and Ravi Chugh (2020). "Program sketching with live bidirectional evaluation." In: *Proceedings of the ACM on Programming Languages* 4.ICFP, pp. 1–29. DOI: 10.1145/3408991.

Luo, Linghui, Julian Dolby, and Eric Bodden (2019). "MagpieBridge: A General Approach to Integrating Static Analyses into IDEs and Editors (Tool Insights Paper)." In: *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom*. Ed. by Alastair F. Donaldson. Vol. 134. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. DOI: 10.4230/lipics.ecoop.2019.21.

Maalej, Walid, Rebecca Tiarks, Tobias Roehm, and Rainer Koschke (2014). "On the Comprehension of Program Comprehension." In: *ACM Transactions on Software Engineering and Methodology* 23.4, pp. 1–37. DOI: 10.1145/2622669.

McBride, Conor (2000). "Dependently typed functional programs and their proofs." PhD thesis. University of Edinburgh, UK. URL: https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.561753.

McBride, Conor (2005). "Epigram: Practical Programming with Dependent Types." In: *Advanced Functional Programming, 5th International School, AFP 2004, Tartu, Estonia, August 14-21, 2004, Revised Lectures*. Ed. by Varmo Vene and Tarmo Uustalu. Vol. 3622. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 130–170. DOI: 10.1007/11546382_3.

McBride, Conor and James McKinna (2004). "The view from the left." In: *Journal of Functional Programming* 14.1, pp. 69–111. DOI: 10.1017/s0956796803004829.

Meijer, Erik (2011). "The world according to LINQ." In: *Communications of the ACM* 54.10, pp. 45–51. DOI: 10.1145/2001269.2001285.

Mens, T. and T. Tourwe (2004). "A Survey of Software Refactoring." In: *IEEE Transactions on Software Engineering* 30.2, pp. 126–139. DOI: 10.1109/tse.2004.1265817.

Mens, Tom, Serge Demeyer, Bart Du Bois, Hans Stenten, and Pieter Van Gorp (2003). "Refactoring: Current Research and Future Trends." In: *Electronic Notes in Theoretical Computer Science* 82.3, pp. 483–499. DOI: 10.1016/s1571-0661(05)82624-6.

Mernik, Marjan, Jan Heering, and Anthony M. Sloane (2005). "When and how to develop domain-specific languages." In: *ACM Computing Surveys* 37.4, pp. 316–344. DOI: 10.1145/1118890.1118892.

Microsoft (2025a). *Visual Studio Code Programming Languages*. URL: https://code.visualstudio.com/docs/languages/overview (visited on 12/15/2025).

Microsoft (2025b). *Visual Studio Technologies*. URL: https://visualstudio.microsoft.com/vs/features/?tab=technologies (visited on 12/15/2025).

Molendijk, Thijs (2022). "Dynamix: A Domain-Specific Language for Dynamic Semantics." MA thesis. Delft University of Technology. URL: https://resolver.tudelft.nl/uuid:8653ab24-a782-41f0-aefc-6b1c8d9a37d5.

Møller, Anders and Michael I. Schwartzbach (2020). *Static Program Analysis*. Department of Computer Science, Aarhus University. URL: https://cs.au.dk/~amoeller/spa/.

Murphy, G. C., M. Kersten, and L. Findlater (2006). "How Are Java Software Developers Using the Eclipse IDE?" In: *IEEE Software* 23.4, pp. 76–83. DOI: 10.1109/ms.2006.105.

Neron, Pierre, Andrew Tolmach, Eelco Visser, and Guido Wachsmuth (2015). "A Theory of Name Resolution." In: *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. Ed. by Jan Vitek. Vol. 9032. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 205–231. DOI: 10.1007/978-3-662-46669-8_9.

Norell, Ulf (2007). "Towards a practical programming language based on dependent type theory." PhD thesis. SE-412 96 Göteborg, Sweden: Department of Computer Science and Engineering, Chalmers University of Technology. URL: https://www.cse.chalmers.se/~ulfn/papers/thesis.html.

Omar, Cyrus, Ian Voysey, Ravi Chugh, and Matthew A. Hammer (2019). "Live functional programming with typed holes." In: *Proceedings of the ACM on Programming Languages* 3.POPL, pp. 1–32. DOI: 10.1145/3290327.

Omar, Cyrus, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer (2017). "Hazelnut: a bidirectionally typed structure editor calculus." In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. Ed. by Giuseppe Castagna and Andrew D. Gordon. Vol. 52. Association for Computing Machinery (ACM), pp. 86–99. DOI: 10.1145/3093333.3009900.

Omar, Cyrus, Ian Voysey, Michael Hilton, Joshua Sunshine, Claire Le Goues, Jonathan Aldrich, and Matthew A. Hammer (2017). "Toward Semantic Foundations for Program Editors." In: *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA*. Ed. by Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi. Vol. 71. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. DOI: 10.4230/lipics.snapl.2017.11.

Opdyke, William F. (1992). "Refactoring Object-Oriented Frameworks." PhD thesis. Urbana-Champaign, IL, USA: University of Illinois. URL: https://www.laputan.org/pub/papers/Opdyke-Thesis.pdf.

Pacak, André and Sebastian Erdweg (2019). "Generating incremental type services." In: *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2019, Athens, Greece, October 20-22, 2019*. Ed. by Oscar Nierstrasz, Jeff Gray, and Bruno C. D. S. Oliveira. ACM, pp. 197–201. DOI: 10.1145/3357766.3359534.

Padhye, Rohan, Koushik Sen, and Paul N. Hilfinger (2019). "ChocoPy: A Programming Language for Compilers Courses." In: *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E*. SPLASH-E 2019. New York, NY, USA: ACM, pp. 41–45. DOI: 10.1145/3358711.3361627.

Palsberg, Jens and Michael I. Schwartzbach (1994). *Object-oriented type systems*. Wiley. ISBN: 978-0-471-94128-6.

Parr, T. J., H. G. Dietz, and W. E. Cohen (1992). "PCCTS reference manual: version 1.00." In: *ACM SIGPLAN Notices* 27.2, pp. 88–165. DOI: 10.1145/130973.130980.

Pelsmaeker, Daniel A. A. (2018). "Portable Editor Services." MA thesis. Delft University of Technology. URL: https://resolver.tudelft.nl/uuid:c8b554de-bcb6-4896-a9bf-c03cca37e344.

Pelsmaeker, Daniel A. A., Hendrik van Antwerpen, Casper Bach, and Eelco Visser (2022a). *Artifact for "Language-Parametric Static Semantic Code Completion"*. DOI: 10.5281/zenodo.6367565.

Pelsmaeker, Daniel A. A., Hendrik van Antwerpen, Casper Bach, and Eelco Visser (2022b). "Language-Parametric Static Semantic Code Completion." In: *Proceedings of the ACM on Programming Languages* 6.OOPSLA1, pp. 1–30. DOI: 10.1145/3527329.

Pelsmaeker, Daniel A. A., Hendrik van Antwerpen, and Eelco Visser (2019). "Towards Language-Parametric Semantic Editor Services Based on Declarative Type System Specifications (Brave New Idea Paper)." In: *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom*. Ed. by Alastair F. Donaldson. Vol. 134. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. DOI: 10.4230/lipics.ecoop.2019.26.

Pelsmaeker, Daniel A. A., Aron Zwaan, Casper Bach, and Arjan J. Mooij (2025a). "Language-Parametric Reference Synthesis." In: *Proceedings of the ACM on Programming Languages* 9.OOPSLA1, pp. 1213–1238. DOI: 10.1145/3720481.

Pelsmaeker, Daniel A. A., Aron Zwaan, Casper Bach, and Arjan J. Mooij (2025b). *Language-Parametric Reference Synthesis (Artifact)*. Version v1. DOI: 10.5281/zenodo.14592164.

Perelman, Daniel, Sumit Gulwani, Thomas Ball, and Dan Grossman (2012). "Type-directed completion of partial expressions." In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*. Ed. by Jan Vitek, Haibo Lin, and Frank Tip. ACM, pp. 275–286. DOI: 10.1145/2254064.2254098.

Pierce, Benjamin C. (2002). *Types and Programming Languages*. Cambridge, Massachusetts: The MIT Press. ISBN: 0-262-16209-1.

Pigott, Diarmuid J. and Bruce M. Axtens (2020). *Online Historical Encyclopaedia of Programming Languages*. URL: https://hopl.info/ (visited on 11/05/2024).

Rascal (2021). *TypePal: Name and Type Analysis Made Easy*. URL: https://docs.rascal-mpl.org/unstable/TypePal (visited on 04/16/2021).

Raychev, Veselin, Martin Vechev, and Eran Yahav (2014). "Code completion with statistical language models." In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. Ed. by Michael F. P. O'Boyle and Keshav Pingali. ACM, pp. 419–428. DOI: 10.1145/2594291.2594321.

Robillard, M. P., W. Coelho, and G. C. Murphy (2004). "How Effective Developers Investigate Source Code: An Exploratory Study." In: *IEEE Transactions on Software Engineering* 30.12, pp. 889–903. DOI: 10.1109/tse.2004.101.

Rouvoet, Arjen, Hendrik van Antwerpen, Casper Bach, Robbert Krebbers, and Eelco Visser (2020). "Knowing when to ask: sound scheduling of name resolution in type checkers derived from declarative specifications." In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA, pp. 1–28. DOI: 10.1145/3428248.

Sasano, Isao and Kwanghoon Choi (2021). "A text-based syntax completion method using LR parsing." In: *Proceedings of the 2021 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM@POPL 2021, Virtual Event, Denmark, January 18-19, 2021*. Ed. by Sam Lindley and Torben Æ. Mogensen. ACM, pp. 32–43. DOI: 10.1145/3441296.3441395.

Schaefer, Max and Oege de Moor (2010). "Specifying and implementing refactorings." In: *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*. Ed. by William R. Cook, Siobhán Clarke, and Martin C. Rinard. Reno/Tahoe, Nevada: ACM, pp. 286–301. DOI: 10.1145/1869459.1869485.

Schäfer, Max, Torbjörn Ekman, and Oege de Moor (2008). "Sound and extensible renaming for Java." In: *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*. Ed. by Gail E. Harris. ACM, pp. 277–294. DOI: 10.1145/1449764.1449787.

Schafer, Max, Andreas Thies, Friedrich Steimann, and Frank Tip (2012). "A Comprehensive Approach to Naming and Accessibility in Refactoring Java Programs." In: *IEEE Transactions on Software Engineering* 38.6, pp. 1233–1257. DOI: 10.1109/tse.2012.13.

Schäfer, Max, Mathieu Verbaere, Torbjörn Ekman, and Oege de Moor (2009). "Stepping Stones over the Refactoring Rubicon – Lightweight Language Extensions to Easily Realise Refactorings." In: *23rd European Conference on Object-Oriented Programming (ECOOP '09)*. Springer Berlin Heidelberg, pp. 369–393. DOI: 10.1007/978-3-642-03013-0_17.

Siegmund, Janet (2016). "Program Comprehension: Past, Present, and Future." In: *Leaders of Tomorrow Symposium: Future of Software Engineering, FOSE@SANER 2016, Osaka, Japan, March 14, 2016*. IEEE, pp. 13–20. DOI: 10.1109/saner.2016.35.

Smits, Jeff and Daniel A. A. Pelsmaeker (2024). "Ordering Rejectable Stacks in SGLR Parsing." In: *Companion Proceedings of the 2024 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. SPLASH '24. ACM, pp. 54–56. DOI: 10.1145/3689491.3691823.

Söderberg, Emma and Görel Hedin (2011). "Building semantic editors using JastAdd: tool demonstration." In: *Language Descriptions, Tools and Applications, LDTA 2011, Saarbrücken, Germany, March 26-27, 2011. Proceeding*. Ed. by Claus Brabrand and Eric Van Wyk. ACM, pp. 1–6. DOI: 10.1145/1988783.1988794.

Solar-Lezama, Armando (2009). "The Sketching Approach to Program Synthesis." In: *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings*. Ed. by Zhenjiang Hu. Vol. 5904. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 4–13. DOI: 10.1007/978-3-642-10672-9_3.

de Souza Amorim, Luís Eduardo, Sebastian Erdweg, Guido Wachsmuth, and Eelco Visser (2016). "Principled syntactic code completion using placeholders." In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31 - November 1, 2016*. Ed. by Tijs van der Storm, Emilie Balland, and Dániel Varró. ACM, pp. 163–175. DOI: 10.1145/2997364.2997374.

de Souza Amorim, Luís Eduardo and Eelco Visser (2020). "Multi-purpose Syntax Definition with SDF3." In: *Software Engineering and Formal Methods - 18th International Conference, SEFM 2020, Amsterdam, The Netherlands, September 14-18, 2020, Proceedings*. Ed. by Frank S. de Boer and Antonio Cerone. Vol. 12310. Lecture Notes in Computer Science. Springer International Publishing, pp. 1–23. DOI: 10.1007/978-3-030-58768-0_1.

Spolsky, Joel (2004). "The Law of Leaky Abstractions." In: *Joel on Software: And on Diverse and Occasionally Related Matters That Will Prove of Interest to Software Developers, Designers, and Managers, and to Those Who, Whether by Good Fortune or Ill Luck, Work with Them in Some Capacity*. Berkeley, CA: Apress, pp. 197–202. DOI: 10.1007/978-1-4302-0753-5_26.

Stack Overflow (2025). *Stack Overflow Developer Survey 2025*. URL: https://survey.stackoverflow.co/2025/ (visited on 12/15/2025).

Steimann, Friedrich (2018). "Constraint-Based Refactoring." In: *ACM Transactions on Programming Languages and Systems* 40.1, pp. 1–40. DOI: 10.1145/3156016.

Steimann, Friedrich, Marcus Frenkel, and Markus Voelter (2017). "Robust projectional editing." In: *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, Vancouver, BC, Canada, October 23-24, 2017*. Ed. by Benoît Combemale, Marjan Mernik, and Bernhard Rumpe. ACM, pp. 79–90. DOI: 10.1145/3136014.3136034.

Steindorfer, Michael (2017). "Efficient Immutable Collections." PhD thesis. Universiteit van Amsterdam. URL: https://michael.steindorfer.name/publications/phd-thesis-efficient-immutable-collections.pdf.

Strachey, Christopher (1973). *Tech. Mono. PRG-10: The varieties of programming language*. Oxford University Computing Laboratory, Programming Research Group.

Tip, Frank (2007). "Refactoring Using Type Constraints." In: *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings*. Ed. by Hanne Riis Nielson and Gilberto Filé. Vol. 4634. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 1–17. DOI: 10.1007/978-3-540-74061-2_1.

Tip, Frank, Robert M. Fuhrer, Adam Kieżun, Michael D. Ernst, Ittai Balaban, and Bjorn De Sutter (2011). "Refactoring using type constraints." In: *ACM Transactions on Programming Languages and Systems* 33.3, pp. 1–47. DOI: 10.1145/1961204.1961205.

Visser, Eelco (2001). "Stratego: A Language for Program Transformation Based on Rewriting Strategies System Description of Stratego 0.5." In: *Rewriting Techniques and Applications, 12th International Conference, RTA 2001, Utrecht, The Netherlands, May 22-24, 2001, Proceedings*. Ed. by Aart Middeldorp. Vol. 2051. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 357–361. DOI: 10.1007/3-540-45127-7_27.

Visser, Eelco, Zine-el-Abidine Benaissa, and Andrew Tolmach (1998). "Building Program Optimizers with Rewriting Strategies." In: *Proceedings of the third ACM SIGPLAN international conference on Functional programming*. Ed. by Matthias Felleisen, Paul Hudak, and Christian Queinnec. Baltimore, Maryland, United States: ACM, pp. 13–26. DOI: 10.1145/289423.289425.

Visser, Eelco, Guido Wachsmuth, Andrew Tolmach, Pierre Neron, Vlad Vergu, Augusto Passalaqua, and Gabrieël Konat (2014). "A Language Designer's Workbench: A One-Stop-Shop for Implementation and Verification of Language Designs." In: *Onward! 2014, Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, part of SPLASH '14, Portland, OR, USA, October 20-24, 2014*. Ed. by Andrew P. Black, Shriram Krishnamurthi, Bernd Bruegge, and Joseph N. Ruskiewicz. ACM, pp. 95–111. DOI: 10.1145/2661136.2661149.

Vollebregt, Tobi, Lennart C. L. Kats, and Eelco Visser (2012). "Declarative specification of template-based textual editors." In: *International Workshop on Language Descriptions, Tools, and Applications, LDTA '12, Tallinn, Estonia, March 31 - April 1, 2012*. Ed. by Anthony Sloane and Suzana Andova. ACM, pp. 1–7. DOI: 10.1145/2427048.2427056.

Völter, Markus (2011). "Language and IDE Development, Modularization and Composition with MPS." In: *Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2011*. LNCS. Springer. DOI: 10.1007/978-3-642-35992-7_11.

Ward, Martin P. (1994). "Language-Oriented Programming." In: *Foundations of F#* 15.4, pp. 271–297. DOI: 10.1007/978-1-4302-0358-2_11.

Xtext Team (2021). *Xtext*. URL: https://www.eclipse.org/Xtext/ (visited on 04/16/2021).

Zwaan, Aron and Hendrik van Antwerpen (2023). "Scope Graphs: The Story so Far." In: *Eelco Visser Commemorative Symposium, EVCS 2023, April 5, 2023, Delft, The Netherlands*. Ed. by Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann. Vol. 109. OASIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. DOI: 10.4230/oasics.evcs.2023.32.

Zwaan, Aron and Casper Bach (2024). "Defining Name Accessibility Using Scope Graphs." In: *38th European Conference on Object-Oriented Programming, ECOOP 2024, September 16-20, 2024, Vienna, Austria*. Ed. by Jonathan Aldrich and Guido Salvaneschi. Vol. 313. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. DOI: 10.4230/lipics.ecoop.2024.47.

Abbreviations

ANTLR

Another Tool for Language Recognition (Parr, Dietz, and Cohen, 1992)

API

application programming interface

AST

abstract syntax tree

BASIC

Beginners' All-purpose Symbolic Instruction Code

BNF

Backus–Naur form

COBOL

Common Business-Oriented Language

CSX

Configuration Space Exploration language

DSL

domain-specific programming language

FGJ

Featherweight Generic Java (Igarashi, Pierce, and Wadler, 2001)

GPL

general-purpose programming language

HTML

Hypertext Markup Language

IDE

Integrated Development Environment

JVM

Java Virtual Machine

LaTeX

Lamport's TeX (Lamport, 1989)

LINQ

Language Integrated Query

LSP

Language Server Protocol

NaBL

name-binding language

RAG

reference attribute grammar

REPL

read–eval–print loop

RPE

robust projectional editor

SDF3

Syntax Definition Formalism 3 (Vollebregt, Kats, and Visser, 2012)

VHDL

Very High Speed Integrated Circuit Hardware Description Language

Curriculum Vitæ

Daniel A. A. Pelsmaeker

now	REIT Research Software Engineer
2024	*Delft University of Technology* in Delft
2024	PhD Computer Science
2018	*Delft University of Technology* in Delft
2024	CSETT Teaching Team Software Engineer
2018	*Delft University of Technology* in Delft
2018	MSc Computer Science
2011	*Delft University of Technology* in Delft
2011	BSc Computer Science
2008	*Delft University of Technology* in Delft
2007	BSc Built Environment (with 1st phase Architecture)
2003	*Hanze University of Applied Sciences* in Groningen
2003	HAVO Senior General Secondary Education
1998	*Ommelander College* in Appingedam
	Dollard College in Woldendorp
1986	Born on 17 July
	in Enschede, the Netherlands

List of Publications

10. ***Daniel A. A. Pelsmaeker***, *Aron Zwaan, Casper Bach, and Arjan J. Mooij*: Language-Parametric Reference Synthesis. OOPSLA 2025. `10.1145/3720481`

9. ***Daniel A. A. Pelsmaeker***, *Aron Zwaan, Casper Bach, and Arjan J. Mooij*: Language-Parametric Reference Synthesis (Artifact). Zenodo 2025. `10.5281/zenodo.14592164`

8. *Jeff Smits, and* ***Daniel A. A. Pelsmaeker***: Ordering Rejectable Stacks in SGLR Parsing. SPLASH 2024. `10.1145/3689491.3691823`

7. *Danny Groenewegen, Elmer van Chastelet, Max de Krieger, and* ***Daniel A. A. Pelsmaeker***, *Craig Anslow*: Conf Researchr: A Domain-Specific Content Management System for Managing Large Conference Websites. EVCS 2023. `10.4230/oasics.evcs.2023.12`

6. *Danny Groenewegen, Elmer van Chastelet, Max de Krieger, and* ***Daniel A. A. Pelsmaeker***: Eating Your Own Dog Food: WebDSL Case Studies to Improve Academic Workflows. EVCS 2023. `10.4230/oasics.evcs.2023.13`

5. ***Daniel A. A. Pelsmaeker***, *Hendrik van Antwerpen, Casper Bach, and Eelco Visser*: Language-Parametric Static Semantic Code Completion. OOPSLA 2022. `10.1145/3527329`

4. ***Daniel A. A. Pelsmaeker***, *Hendrik van Antwerpen, Casper Bach, and Eelco Visser*: Artifact for "Language-Parametric Static Semantic Code Completion". Zenodo 2022. `10.5281/zenodo.6367565`

3. ***Daniel A. A. Pelsmaeker***, *Hendrik van Antwerpen, and Eelco Visser*: Towards Language-Parametric Semantic Editor Services Based on Declarative Type System Specifications (Brave New Idea Paper). ECOOP 2019. `10.4230/lipics.ecoop.2019.26`

2. ***Daniel A. A. Pelsmaeker***: Portable Editor Services (Master's thesis). TU Delft 2018. `https://resolver.tudelft.nl/uuid:c8b554de-bcb6-4896-a9bf-c03cca37e344`

1. *Tim van der Lippe, Thomas Smith,* ***Daniel A. A. Pelsmaeker***, *and Eelco Visser*: A scalable infrastructure for teaching concepts of programming languages in Scala with WebLab: an experience report. SCALA 2016. `10.1145/2998392.2998402`

Dissertation Series

Titles in the IPA Dissertation Series since 2023

S. Kochanthara. *A Changing Landscape: On Safety & Open Source in Automated and Connected Driving*. Faculty of Mathematics and Computer Science, TU/e. 2023-01

L.M. Ochoa Venegas. *Break the Code? Breaking Changes and Their Impact on Software Evolution.* Faculty of Mathematics and Computer Science, TU/e. 2023-02

N. Yang. *Logs and models in engineering complex embedded production software systems.* Faculty of Mathematics and Computer Science, TU/e. 2023-03

J. Cao. *An Independent Timing Analysis for Credit-Based Shaping in Ethernet TSN.* Faculty of Mathematics and Computer Science, TU/e. 2023-04

K. Dokter. *Scheduled Protocol Programming.* Faculty of Mathematics and Natural Sciences, UL. 2023-05

J. Smits. *Strategic Language Workbench Improvements.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2023-06

A. Arslanagić. *Minimal Structures for Program Analysis and Verification.* Faculty of Science and Engineering, RUG. 2023-07

M.S. Bouwman. *Supporting Railway Standardisation with Formal Verification.* Faculty of Mathematics and Computer Science, TU/e. 2023-08

S.A.M. Lathouwers. *Exploring Annotations for Deductive Verification.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2023-09

J.H. Stoel. *Solving the Bank, Lightweight Specification and Verification Techniques for Enterprise Software.* Faculty of Mathematics and Computer Science, TU/e. 2023-10

D.M. Groenewegen. *WebDSL: Linguistic Abstractions for Web Programming.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2023-11

D.R. do Vale. *On Semantical Methods for Higher-Order Complexity Analysis.* Faculty of Science, Mathematics and Computer Science, RU. 2024-01

M.J.G. Olsthoorn. *More Effective Test Case Generation with Multiple Tribes of AI.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-02

B. van den Heuvel. *Correctly Communicating Software: Distributed, Asynchronous, and Beyond.* Faculty of Science and Engineering, RUG. 2024-03

H.A. Hiep. *New Foundations for Separation Logic.* Faculty of Mathematics and Natural Sciences, UL. 2024-04

C.E. Brandt. *Test Amplification For and With Developers.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-05

J.I. Hejderup. *Fine-Grained Analysis of Software Supply Chains.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-06

J. Jacobs. *Guarantees by construction.* Faculty of Science, Mathematics and Computer Science, RU. 2024-07

O. Bunte. *Cracking OIL: A Formal Perspective on an Industrial DSL for Modelling Control Software.* Faculty of Mathematics and Computer Science, TU/e. 2024-08

R.J.A. Erkens. *Automaton-based Techniques for Optimized Term Rewriting.* Faculty of Mathematics and Computer Science, TU/e. 2024-09

J.J.M. Martens. *The Complexity of Bisimilarity by Partition Refinement.* Faculty of Mathematics and Computer Science, TU/e. 2024-10

L.J. Edixhoven. *Expressive Specification and Verification of Choreographies.* Faculty of Science, OU. 2024-11

J.W.N. Paulus. *On the Expressivity of Typed Concurrent Calculi.* Faculty of Science and Engineering, RUG. 2024-12

J. Denkers. *Domain-Specific Languages for Digital Printing Systems.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-13

L.H. Applis. *Tool-Driven Quality Assurance for Functional Programming and Machine Learning.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-14

P. Karkhanis. *Driving the Future: Facilitating C-ITS Service Deployment for Connected and Smart Roadways.* Faculty of Mathematics and Computer Science, TU/e. 2024-15

N.W. Cassee. *Sentiment in Software Engineering.* Faculty of Mathematics and Computer Science, TU/e. 2024-16

H. van Antwerpen. *Declarative Name Binding for Type System Specifications.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2025-01

I.N. Mulder. *Proof Automation for Fine-Grained Concurrent Separation Logic.* Faculty of Science, Mathematics and Computer Science, RU. 2025-02

T.S. Badings. *Robust Verification of Stochastic Systems: Guarantees in the Presence of Uncertainty.* Faculty of Science, Mathematics and Computer Science, RU. 2025-03

A.M. Mir. *Machine Learning-assisted Software Analysis.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2025-04

L.T. Vinkhuijzen. *Data Structures for Quantum Circuit Verification and How To Compare Them.* Faculty of Mathematics and Natural Sciences, UL. 2025-05

D. van der Wal. *What is the Point? Single-Input-Change Testing a EULYNX Controller.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2025-06

A. Rosset. *Uniform Monad Presentations and Graph Quasitoposes.* Faculty of Sciences, Department of Computer Science, VU. 2025-07

L. Guo. *Higher-Order Termination with Logical Constraints.* Faculty of Science, Mathematics and Computer Science, RU. 2025-08

D.A. Manrique Negrin. *A Model Orchestra in Digital Twins: A Model-Driven Approach to Integration and Orchestration.* Faculty of Mathematics and Computer Science, TU/e. 2025-09

C.A. Esterhuyse. *Specification-Centric Multi-Agent Systems.* Faculty of Science, UvA. 2025-10

H.M. Muctadir. *Consistency Matters: Building Consistent Digital Twin Virtual Entities.* Faculty of Mathematics and Computer Science, TU/e. 2025-11

A. Stramaglia. *Model Checking Machine-Control Applications.* Faculty of Mathematics and Computer Science, TU/e. 2025-12

M. Saeedi Nikoo. *Supporting business process management: clone detection and recommendation techniques.* Faculty of Mathematics and Computer Science, TU/e. 2025-13

R.B. Rubbens. *Bridging the Implementation Gap: Advances in Model-Based Concurrent Program Verification.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2025-14

F.I. van der Berg. *DMC Model Checker: Delta-Driven Variable-Length Next-State Generation via Recursive Compression.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2025-15

D.G. Sprokholt. *Correct Translation between Weak Memory Model Architectures.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2025-16

B. Kohlen. *Verified & Fast Probabilistic Verification: Stepwise Refinement of Probabilistic Model Checking into an IEEE 754 Floating-Point Implementation.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2026-01

D. Frolich. *Exploratory Language Development.* Faculty of Science, UvA. 2026-02

L.B. van den Haak. *Verifying Optimised Parallel Code.* Faculty of Mathematics and Computer Science, TU/e. 2026-03

A. Papotti. *Human Acceptance of Machine Generated Code Security Analysis.* Faculty of Sciences, Department of Computer Science, VU. 2026-04

N. Doorn. *O bug, where art thou? Improving software testing education.* Faculty of Science, OU. 2026-05

C.R. van der Rest. *Reusable Programming Language Components.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2026-06

T.T.P. Franken. *A Parallel Programming Perspective – Designing AuDaLa and AuDaLogic.* Faculty of Mathematics and Computer Science, TU/e. 2026-07

T.J. Smeding. *Reverse-mode Automatic Differentiation for Functional Array Languages.* Faculty of Science, UU. 2026-08

S.A. Rukmono. *Towards Automated Software Architecture Explanation: An AI-enabled method and industrial evaluation.* Faculty of Mathematics and Computer Science, TU/e. 2026-09

R.T.C. Turkenburg. *Comparing and Constructing (Co)Inductive Predicates.* Faculty of Science, Mathematics and Computer Science, RU. 2026-10

P.B.H. Tasche. *Deductive Verification Techniques for Embedded Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2026-11

W.B. Mbaka. *An Empirical Investigation Into Human Judgments of Security Threats.* Faculty of Sciences, Department of Computer Science, VU. 2026-12

S. Srinivasan. *Taming Uncertainty, Enabling Predictability: Systematic Timing Analysis of Real-Time Systems Across Computation and Communication.* Faculty of Mathematics and Computer Science, TU/e. 2026-13

S.O. Brand. *Symbolic Algorithms for the Analysis of Quantitative Systems.* Faculty of Mathematics and Natural Sciences, UL. 2026-14

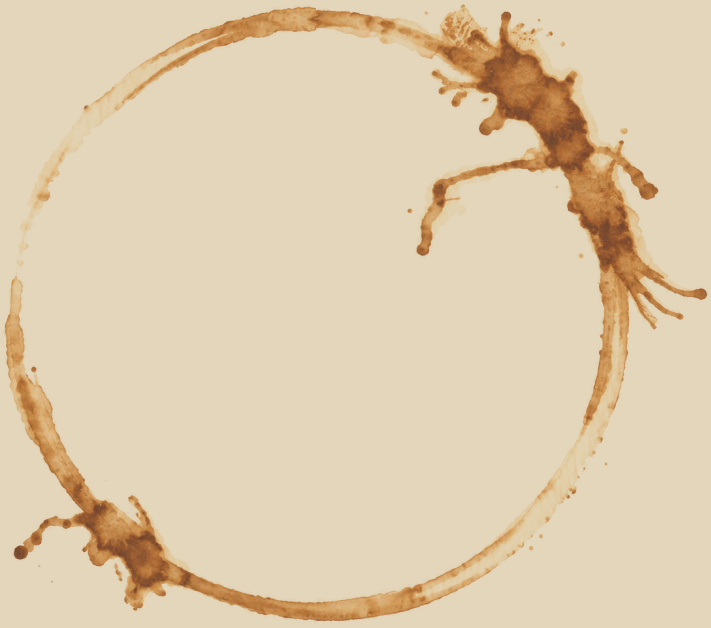
D. Engel. *Understanding Binaries, Correctly: Decidability and Formal Verification in Binary Analysis.* Faculty of Science, OU. 2026-15

K. Hagens. *Rewriting Induction for Higher-Order Constrained Rewriting.* Faculty of Science, Mathematics and Computer Science, RU. 2026-16

S. Khatami. *Understanding Software Quality Assurance in Open-Source Communities: Awareness, Adoption, and Tool Utilization.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2026-17

R. Soltani. *Defense Against the Dark Arts: Formal Semantics and Quantitative Analysis of Attacks, Failures, and Defenses.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2026-18

D.A.A. Pelsmaeker. *Language-Parametric Editor Services from Language Specifications.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2026-19



9 789465 630731